RL-TR-97-36
Final Technical Report
June 1997

OBJECT-ORIENTED METHODS AND TOOLS

Syracuse University

Umesh Bellur, Gary Craig, Douglas Lea, and Kenn Shunk

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Rome, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-97-36 has been reviewed and is approved for publication.

APPROVED: Anthony M. Newton

ANTHONY M. NEWTON
Project Engineer

FOR THE COMMANDER: John A Graniero

JOHN A. GRANIERO, Chief Scientist
Command, Control & Communications

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL/C3AB, 525 Brooks Rd, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE Jun 97	3. REPORT TYPE AND DATES COVERED FINAL Apr 93 - Jun 94

4. TITLE AND SUBTITLE

OBJECT-ORIENTED METHODS AND TOOLS

5. FUNDING NUMBERS

C - F30602-93-C-0108
PE - 63728F
PR - 2530
TA - 01
WU - P2

6. AUTHOR(S)

Umesh Bellur, Gary Craig, Douglas Lea, Kenn Shunk

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Syracuse University
Syracuse, NY

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Rome Laboratory (C3AB)
525 Brooks Rd.
Rome, NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

RL-TR-97-36

11. SUPPLEMENTARY NOTES

Rome Laboratory Project Engineer: Anthony M. Newton/C3AB/ (315) 330-3097

12a. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This project seeks to address the development of software systems targeted for heterogeneous distributed systems. In particular, this work is aimed at the discovery and construction of software development methods, corresponding software tools (programs), and consistent infrastructure that improve the quality and productivity of object-oriented (OO) software engineering. DIAMONDS is the core project of this effort. The two major goals of the DIAMONDS project are: 1) the specification and validation of a sound object-oriented software development methodology for heterogeneous distributed processing, and 2) the development and support of an object (software) composition model which permits a system to effectively exploit an application's inherent concurrence with efficient communication and management overhead. The overall goal of DIAMONDS is to produce a methodology and supporting infrastructure such that the mapping of an application to system resources dynamically reflects algorithmic parallelism, computational needs, and the current state of the system. These goals, when met, greatly enhance the ability of a software product to be effectively deployed in a wide variety of system settings.

14. SUBJECT TERMS

Distributed systems, object oriented, software methods, software tools, operating system, distributed computing, DCE.

15. NUMBER OF PAGES
142

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR

Contents

List of Tables

List of Figures

Chapter 1

Problem Statement and Approach

This project seeks to address the development of software systems targeted for heterogeneous distributed systems. In particular, this work is aimed at the discovery and construction of software development methods, corresponding software tools (programs), and consistent infrastructure that improve the quality and productivity of object-oriented (OO) software engineering. Object-oriented methods have been widely embraced by industrial software development efforts. While already successful, OO methods and their support infrastructure still require further study and exploration in order to make good on their full potential.

DIAMONDS is the core project of this effort. The two major goals of the DIAMONDS project are:

1. the specification and validation of a sound object-oriented software development methodology for heterogeneous distributed processing, and

2. the development and support of an object (software) composition model which permits a system to effectively exploit an application's inherent concurrency with efficient communication and management overhead.

A distributed system is comprised of networks of heterogeneous processors which serve as a pool of computational resources. Distributed software systems are difficult to specify, design, implement and maintain. This task is made more difficult when the target architecture is highly varied or constantly changing, as is the case for heterogeneous distributed systems.

The overall goal of DIAMONDS is to produce a methodology and supporting infrastructure such that the mapping of an application to system resources dynamically reflects algorithmic parallelism, computational needs, and the current state of the system. These goals, when met, greatly enhance the ability of a software product to be effectively deployed in a wide variety of system settings.

The philosophy for accomplishing the goals of DIAMONDS is to construct a development model, a set of tools, and a run-time infrastructure in unison. Without the unique interplay between the components, the opportunity for success is greatly diminished. DIAMONDS seeks to improve the ability to produce effective distributed software. Quality is enhanced by a development methodology providing a smooth transition path from analysis through to implementation. Resource consumption and performance issues are addressed by a comprehensive object composition model and a well matched infrastructure.

The present research plan of approach, places approximately equal emphasis on four main research and development areas, *Models*, *Methods*, *Tools*, and *Infrastructure.* Integrated efforts in these areas are needed in order to improve the quality and productivity of OO software development.

1.1 Plan of Approach

There are four major components of the present research program:

1. *Models.* Development of theoretical models of OO programs that in turn form the basis for software that translates, represents, analyzes, and verifies OO software.

2. *Methods.* Development of prescriptive engineering methodologies mapping the tasks necessary for creating software into practical development strategies; along with development of techniques and idioms to solve common OO design problems.

3. *Tools.* Construction of prototype software tools based on these models and methods, and assessment of their utility.

4. *Infrastructure.* Exploration and development of infrastructure software better supporting these models, methods, and tools.

1.1.1 Abstract Object Models

The most central component of this research program is the development of a tractable model of object-based computation. This is an active area of academic research, integrating diverse work from logic, semantics, concurrent and distributed computation, database, artificial intelligence, and simulation in order to arrive at models that describe their common underpinnings as reflected in object computation. The specific contributions of the proposed research are very much tied to its engineering orientation. We require abstract models that directly support the development of OO tools and infrastructure. Such models must unambiguously define a "virtual machine" lending itself for use in the semantic representation and analysis of OO programs and systems.

Initial modeling and analysis considerations central to our work are described in [Lea91]. A corresponding design notation (odl) and language is presented both in [dCLF93] and [Lea94].

1.1.2 Methodology

Object-oriented software engineering (OO-SE) practices are rapidly becoming the methods of choice across a wide range of software development efforts. For the most part, the trend has been for OO approaches to infiltrate up from programming techniques, to design strategies, to analysis and modelling frameworks, and finally to entire software development process models. Correspondingly, "pure" OO approaches are being applied to increasingly larger and more varied engineering efforts. Until recently, bits and pieces of OO methods were most typically somehow fit into other development approaches. However, the advent of 100% OO-SE is already upon us [Rea91, Jea92, dCLF93, Boo94, Cea94].

Despite this, OO-SE methodology is not as firmly understood as it ought to be. There are two general kinds of questions: (1) How do you characterize and differentiate "good" from "bad" OO-SE *processes* (i.e., the series of steps taken from conception to delivery of software systems)? (2) How do you characterize and differentiate "good" from "bad" individual OO analysis, design, implementation, testing, etc., methods?

Process Issues

Even within a traditionally-motivated framework which separates analysis, design, and programming related efforts, there are a number of differences between prototypical OO versus "structured" development processes. These stem from essential properties of the OO paradigm, including:

- *Modeling.* OO requirements, specifications, designs, and even programs are generally viewed by developers as models (of varying degrees of abstraction and precision) of particular domains and applications. Hence the artifacts of development phases not only lead to the construction of executable software, but also represent understandable models of underlying declarative and computational frameworks.

- *Classes.* Classes form the "natural" focal points for development. Class-based strategies entail component-centered versus application-centered practices, the use of hierarchical techniques (especially subclassing) to relate and extend components, and placing software reuse issues prominently in all phases of development.

- *Continuity.* OO analysts, designers, database engineers, programmers, and other developers all employ approximately the same vocabulary, concepts, methods, and idioms. This continuity enables and rationalizes well-known OO practices including iterative refinement, early prototyping, and mixed top-down and bottom-up approaches.

OO Analysis (OOA) models are "naturally" highly parallel – the most effective declarative OOA practices are those that assume that every object in a system is a computational agent (virtual processor). Both hardware technology and the sheer size of OO systems increasingly require that implementations also be realized across multiple physical processors. This leads to some rather striking differences in design practices that must be supported within the overall development process. Most notably, rather than "distributing" a design by splitting functionality across processors, OO designers should assume at the outset massive MIMD parallelism. They may then "cluster" and sequentialize objects within physical processes in the nearly-universal case that there are more objects in a system than there are processors.

Methods

The description, evaluation, prescription, compilation, and discovery of general analysis, design, and implementation techniques, idioms, and practices is a difficult, sometimes fuzzy, but exceedingly important area of study. A number of quality criteria may be employed, including:

- *Analytic.* Safety, correctness, liveness, determinism efficiency, complexity.

- *Usability and Reusability.* Coupling, cohesion, generality, abstraction, capability for multiple instantiation, extensibility, fitness for use as components, interoperability, support for concurrency, support for multiple interaction and access control protocols, encapsulation, testability.

- *Pragmatic.* Understandability and related human factors, compatibility with other components, conservatism, maintainability, usability at multiple development levels.

Some aspects of particular models, designs, and implementations may be assessed within a uniform OO computational model, as above. Other aspects require more qualititative analysis. However, no evaluation method can automatically derive methods that always generate software entities possessing any given property. But this may be approached by creating notations and tactics that help guarantee at least some aspects of *quality by construction.*

Design tools

As discussed above, we are devising design notations that (1) increase the likelihood that developers produce designs that are "good" by construction; (2) enable designs to be mechanically analyzed. The development of static analysis techniques appropriate for OO designs is among the most central aspects of the proposed research.

A well-developed static analysis framework built upon a uniform abstract object model provides a basis for a large number of related analyses that are currently supported at best by weaker one-shot facilities, if at all. They include:

1. *Development Environment Support.* IsA, Uses, Communicates-with, and other relations among classes.
2. *Dynamic properties.* Liveness, Aliasing, Dispatching safety.
3. *Constraint flow analysis.* Invariants, Preconditions, Postconditions, Type safety.
4. *Optimization.* Customization, Message-splitting, Embedding, Caching.
5. *Distribution.* Messaging, Location, Mobility, Clustering
6. *Metrics.* Complexity, Coupling, Efficiency.
7. *Simulation.* Prototype interpretation and animation.
8. *Code generation.* Translation into multiple target languages.

This specific effort has resulted in a prototype framework which is used to perform analysis for purposes of distribution, optimization, simulation and code generation

1.1.3 Infrastructure Support for OO Distributed Processing

The development of distributed software requires a high level of cooperation between the software development environment (language and tools) and the distributed run-time support services. DIAMONDS is a project whose goal is to investigate this interplay between software design methods and tools and the distributed run-time environment.

DIAMONDS

In DIAMONDS, we are investigating support for distribution via an OO programming model which is inherently distributed with particular emphasis placed on the role of methods and tools. This philosophy for support of distribution stems from the fact that it is unlikely that a developer will achieve a global view of the run-time configuration of a non trivial distributed processing system. It is our view that application partitioning and distribution are tightly bound to the run time state of the system and as such, are difficult to predict during software development. Thus it is important to couple static support for partitioning with dynamic run-time support.

Some of the important attributes and issues within DIAMONDS include:

Programming Model Our basic programming model, which follows directly from the OO models work, is comprised of fine-grained objects, where all objects are *potentially* active. The internal representation of an object is dictated by the *host* architecture on which the object currently resides. There may be several versions of the object's behavior depending on the architectures it may migrate to. The default object communication model is asynchronous invocation. Replies to asynchronous invocations (if required) are by means of "call backs" – returning a reply by invoking one of the client's methods. Synchronous invocation is also supported.

Concurrency Our object model inherently supports fine-grained parallelism. Clustering of objects into coarser units assists in efficiently supporting loosely-coupled systems. This means that both fine and coarse grained parallelism can be expressed and supported. The model also takes into account synchronization issues that crop up when we consider active objects with multiple invocations possible simultaneously.

Object Clusters An object cluster is a logical entity formed by the compiler either through language directives or through static analysis. Clusters are processes serving as repositories for groups of objects. A cluster might be formed to:

- *Optimize communication:* Intra-cluster inter-object communication may be highly optimized.
- *Maximize concurrency:* Objects displaying high degree of cooperative parallelism will be placed in different clusters.
- *Present a collective interface:* A object composition mechanism.

DIAMONDS requires extensive run-time services. These will be provided through a unique blend of what are currently either language specific runtime features or services provided by an operating system.

Late binding of communication structures Depending on whether the communication is of the inter-cluster or intra-cluster variety, dynamic binding of the appropriate communication mechanism will be done to enhance performance.

Mobility In order to deal with dramatic changes in load and network connectivity, clusters may need to migrate.

Resource Management The resource management services are responsible for mapping clusters to physical containers (*Ensembles*), allocation of these physical containers, and the run-time reconfiguration of these entities within the distributed system.

1.1.4 Summary

This report will present the detail efforts of an ongoing research project which seeks to strengthen the ability to develop quality object-oriented software for heterogeneous distributed systems. Such an effort involves a tight interplay between subdiscipline activities in OO softare engineering models, methods, tools and infrastructure.

The remainder of this report is organized as follows: chapter 2 addresses in detail the fine-grained active object model which forms the basis for the design and implementation of concurrent OO software systems; next, chapter 3 presents the fundamental computation model which drives our view of the run-time system; chapter 4 considers the issues involved in applying static analysis to the task of object clustering; chapter 5 considers the run-time infrastructure requirements of DIAMONDS; next chapter 6 outlines the current status of each of our prototype components; and finally, chapter 7 looks forward to current and future work.

Chapter 2

Design and Programming Model

The object-oriented paradigm is evolving from a set of programming techniques to a mature software engineering methodology. A range of OO software development methods represent variations on a common OO systems development paradigm, that may be briefly and informally characterized as follows:

1. Discover some tentative objects needed to fulfill functionality requirements.
2. Model these objects (and/or others discovered in the course of elaboration) and their interactions by assuming that *all* of them are autonomous computational entities (i.e., "active objects").
3. Develop computational designs logically behaving in accord with these models. (There may be almost arbitrarily small differences between this and the previous step.)
4. Map this design onto available physical configurations, software services, tools, and languages.

This model is implicit or explicit in many formal and informal OO system development methods and practices. Such development methods stress autonomy and tend to discourage premature commitment to task-specific control flow. Other notable consequences of this development model include:

- Systems are logically *composed* of many interacting active agents, rather than decomposed into task-oriented processes.

- Design-level objects come in all shapes and sizes, ranging conceptually from tiny (e.g., Integers) to huge (e.g., Banks). However, even "huge" objects tend to be representationally small, since they obtain most functionality by delegating actions to other potentially autonomous objects.

- The decision of whether particular objects are to be implemented in an "active" versus "passive" manner is (or would be, given appropriate support) a relatively late development issue. At the level of software analysis and design, there are often no meaningful intrinsic differences between objects that may be implemented "passively" versus those that may be implemented in stand-alone processes.

- Protocols involving event propagation, callbacks, multicast, and explicit routing are natural components of many models and designs. For example, model-view-controller style change notifications are usually best construed as asynchronous one-way notifications.

- Typical applications include more state-dependent and state-modifying operations than stateless services.

These features in turn stem from reliance on object models with the following properties, applicable to *any* object, from a simple Boolean, to a BankAccount, to a TextEditor:

1. Every object has a unique identity.
2. Every object is an instance of some class. Classes may be defined as subclasses of others.
3. Instances are constructed and maintained via generator objects.
4. Static object features are defined via attributes, relations, and constraints.
5. Every object is defined as a sequential computational agent, performing at most one operation at any given time.
6. All visible state transitions and operations are defined as *atomic* with respect to external observers.
7. Operations are *guarded* with state-based constraints. Objects are normally defined to contain queues to buffer pending untriggerable messages.
8. All object interaction occurs via one-way (asynchronous) and/or bidirectional (procedural) message passing.

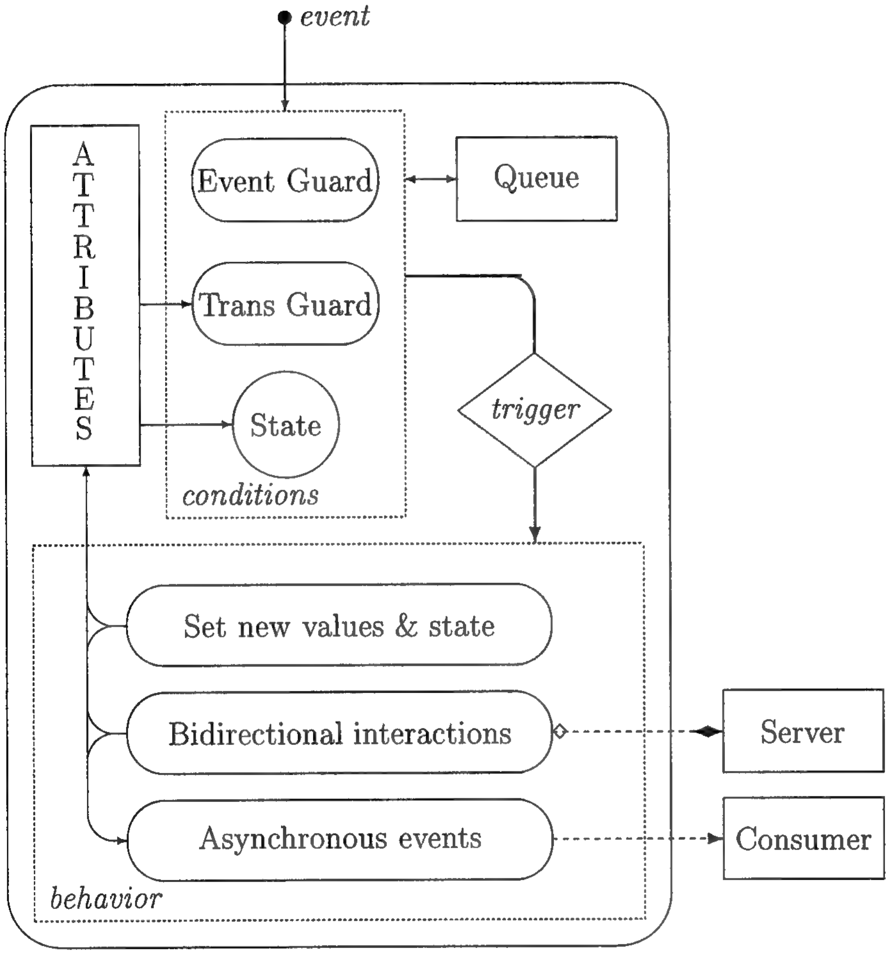

Fig. 1. Basic active object.

Fig. 1 illustrates the fundamental properties of an active object. It has a message handling/dispatch unit, local state, and actions comprising only of changes to local attributes, the raising of additional events, and, for certain objects (unshown here) the construction of new objects.

The resulting model applies independently of granularity issues. "Active" mechanics are sometimes needed even for "small" objects. For DIAMONDS, the model is captured in ODL (*Our Design Language*) which was original devised for use in presenting object-oriented system design concepts in a programming language- and system-independent manner.

ODL is in most respects a typical object-oriented programming language, containing constructs corresponding to the notions of classes, attributes, methods, objects, and links, but geared towards distributed programs. Programs are written by defining classes, operations, and supporting type and execution information.

2.1 Names and Scopes

ODL declarations consist of named classes, parameterized classes, records, fields, slots, and constraints. The constructs `class`, `op`, `fn` and `record` introduce named scopes.

Class declarations may be nested within others. Class names introduced at the same scope level must be unique. Name resolution for embedded classes, fields and slots follows most-closely-nested rules but may be redirected via the *name*`::` operator, where *name* is a named scope introducer. Any name may be qualified as `local`, which disables reference outside of the current scope except within constructor expressions.

The top-level declarations of an entire ODL program are considered to be encased within `class System` ...`end`. Class `System` must contain single user-defined operation `main`. An ODL implementation must support a command equivalent to `(new System(...)).main(...)` that constructs object `system` and initiates execution.

Normally, names must be declared before they are otherwise referenced. However, multiple declarations may be listed in mutually referential fashion by enclosing them within `letrec` ...`end`. Also `generator` statements may mention as-yet undeclared classes.

2.2 Types

There are four kinds of value types: scalars, links, arrays, and records. Values of these types are not themselves objects. A fifth kind, that of code bodies defining slots is a "second-class" type, discussed separately below.

The scalar types are `bool`, `int`, `real`, `char`, `time`, and `blob`. Literals for each type are self-describing. The first four have the usual meanings. Restrictions on the bounds and precision of implementations of these types are permitted but not otherwise defined. The `time` type denotes times. It is left unspecified whether `time` is a discrete or continuous quantity. The type `blob` describes completely opaque values that may have implementation-dependent meaning.

A link is a value denoting the identity of an instance. Link types are introduced via the declaration of classes of the same name. Their subtype structure mirrors the declared inheritance relations among classes. Other scalar types do not obey any subtype structure. They are distinct, incommensurate types in ODL.

Fixed-bound vector values are designated via postfix `[`*capacity*`]`. Elements are referenced using subscripts.

Records are named tuples of values. Records define fields of any of the four kinds of value type. Nested fields are referenced via dot notation. Any field declaration may be qualified as:

`common` The value must be the same across all instances of this record.

`unique` The value must differ across all instances of this record.

`opt` The value need not be present. A record declaration with an `opt` field is considered equivalent to two declarations, one with the field, and one without it. A declaration with two `opt` fields is equivalent to four, and so on. However, syntactically these versions may be described and used as an aggregate. Absence of a value may be indicated via `null` and discerned via `null(`*field*`)`. Access to a nonpresent `opt` value may be trapped as a static program error, dynamic execution error, or left undetected by translators.

All combinations of these qualifiers are semantically meaningful and allowed; for example, even the odd combination of `common` and `unique`, which is an awkward way of saying that only one value is possible.

Record values are expressed by naming the record type followed by a parenthesized list of values for each field. Both positional and named syntax is allowed, but all positional values must precede named ones, fields indicated positionally may not be listed also as named, and names may not be listed more than once. Lack of value for an `opt` field need not be listed at all in the named syntax. For example, all three expressions denote the same value in:

```
class A ... end;
record r(a: int, b: char, c: real, d: opt A);

r(1, '2', 3.0, null)
r(b := '2', a := 1, c := 3.0, d := null)
r(1, '2', c := 3.0)
```

2.3 Classes

Classes describe families of objects, all of which possess the same structural features. Each object is viewed as a map from a unique identity to a set of names. The map is the same for all direct instances of a given class. These names in turn map to *slots*, which may be different for each object. Unlike record fields, slots are computationally defined. There are three kinds of slots:

- One-way operations, expressed as `op m(args)`. One-way operations are those that accept a message and perform some computation independently of the message sender.
- Procedural operations, expressed as `op p(args) : replyName(args)` or `op p(args) resultName : resultType`. The client of a procedural operation waits for the recipient to reply before proceeding.

- Functional operations, expressed as `fn f(args) : fnType`. Functional operations are special side-effect-free forms of procedures. Functions may be defined concretely either via code bodies or via a special form indicating that a single value is "stored".

All slot declarations within a class are processed as if encased within `letrec` ... `end`. Slot names and arguments normally correspond to message names and fields that are accepted by instances of the class. However, internal `local` versions of any of the above may also be declared. Groups of `local` slots may be declared within `locals` ... `end`.

2.3.1 Concrete Definitions

Concrete definitions may be bound to slots via {...}. A special form, <> is used instead to indicate that a slot is bound to a concrete definition only upon construction. It is used only for "stored" functional slots that access a single value that must be intialized upon construction.

Concrete slot definitions may contain the following constructs:

Message Sends: One-way sends of messages to specified recipients.

Procedure Invocations: Blocking call/return style interaction.

Function Invocations: Side-effect-free procedural interaction.

Local Invocations: Internal sequential processing.

Rebindings: Assignment statements rebind non-`fixed` stored slots to different values via `f :=` *exp*.

Locals: `local` declarations of new transient slots maintained only within an operation.

Control flow: `if` and `while` statements controlled by boolean value expressions.

Message Sends

Computation is based on message passing. Objects are autonomous single-threaded computational agents that send messages listed within operations corresponding to messages from other objects. A *message* is a record that corresponds to a declared operation. `ODL` `op` declarations define records with names identical to operation names, and fields corresponding to argument lists. There are five phases in any act of message passing:

Invocation. An invocation listed in the concrete definition of a sender is issued.

Reception. ODL does not specify the mechanisms controlling how messages are issued and received by objects except in the assumption that they do not interfere with explicitly defined processing. Synchronicity between sending and receiving a message is neither required nor precluded.

Binding. A message is linked with a corresponding operation or version of an operation. Linkage may be determined either statically or dynamically. (Run-time binding is necessary when there are argument-based guard conditions.) Binding failures cannot occur in correct programs.

Acceptance. An accepted message is "consumed", and causes the triggering of an operation when its guard has cleared.

Execution. Computation proceeds by noninterruptibly processing all actions defined in the corresponding concrete operation.

The simplest form of a concrete operation is a sequence of one-way message sends, for example, { `a.m1(x); b.m2(y, z); ...`}. Invocations are defined by by naming them and providing field values along with a recipient designation. Instead of recipient prefixing, ODL messages may be invoked with an indication of a *class* of receivers, via *className$message*. Stylistically, this form is useful for stateless services for which the identity of the recipient cannot matter. The run-time system is free to select any instance of the indicated class (or any subclass thereof) to receive the message. ODL does not prescribe a particular translation mechanism. Several are available. For example, because all occurrences of $ are statically determinable before execution, the system may construct a pool of such objects upon initialization and translate all invocations to normal prefixed form.

Procedures

In the base syntax of procedural interaction, result-bearing operations define message records for returned values, and clients define corresponding operations to accept them:

```
class A op m1(x: T) : result(b: B) ... end end;

class Auser ...
  op calla(a: A) {
    catch a.m1(x)
      op result(b: B) { b.m2(y, z); ... }
    end  }
end
```

Here, the sender enters a state in which its only next action is to receive a corresponding reply. A `catch` clause introduces one or more transiently available operations that accept replies from servers. Multiple named result messages and catches are also allowed. The names of the client operations must match those listed for the server. Translators must arrange that result operations be caught only when objects are in the required state.

A server object invokes these transient messages by name:

```
class A op m1(x: T) : result(b: B) { ...; result(new B); } end;
```

The recipient of the reply is left implicit. This logically requires that sender identity be transmitted as an implicit argument in all procedural messages. However, in ODL the sender identity is not otherwise accessible to the server. (Unless, of course, a sender field is explicitly added to the operation signature and used in the desired ways.) A server may perform additional actions after issuing a reply.

More conventional looking procedural forms are also supported, via via anonymous return messages. For example:

```
  op a : i: int;
  op b : ()
```

This declares the result of a as an anonymous record with single field `i` and the result of b as an anonymous, fieldless record. Each anonymous record is considered to have a different name. Anonymous return messages are sent via `reply`:

```
class A2 op m1(x: T) : B {  ...; reply b; } end;

class Auser  ...
  op calla2(a: A2) { local b :B := a.m1(x); b.m2(y, z);  }
end
```

Here, the anonymous `catch` may be elided, and the reply used directly in a procedural fashion.

Simple blocking procedural interaction is the only two-way protocol natively supported in ODL. Others may be defined through combinations of one-way sends and object constructions. For example, a *future* may be defined via the construction of a helper object to wait out a procedure.

Functions

Functional operations have a restricted form. They are defined as single expressions using the value expression sublanguage described in section 2.3.2. Translators are required to transform functional expressions into procedural computations (possibly involving new independent, unreachable objects) that cannot interfere with other operations and objects.

Stored functions are yet further restricted. They may be defined only via <>, indicating that a stored value be attached upon construction, retrieved upon access, and possibly rebound in the course of other operations. Stored links may also be qualified as `packed`. This is a hint to the translator that the object referenced by the slot should be embedded within the representation of the host object.

The base form of stored values is restricted to link values, not other types. The reflects an underlying object model in which state varies only as a function of connections among objects. Other forms may be implemented with the help of instances of elementary predefined classes. However, ODL programmers are not required to do so themselves. Translators may mechanically reduce them to base form. A stored value denoting a non-link type may be translated to one holding a link to an instance of a predefined class, where value accesses are forwarded to these objects. Value rebindings may be translated either to link rebindings of new objects with the required initial values or to `set` operations on the exisiting objects, or any other technique, at the discretion of the translator. Bindings of the form `l := null` for `opt` slots are handled similarly.

Local Operations

ODL `local` operation invocations are not received as messages. They are sequentially executed in the course of performing other actions. To avoid the need for redundant declaration, a *local* version of each functional non-local operation is automatically constructed if not otherwise present. Local operations must be invoked without a recipient prefix. (In contrast non-local self-invocations must be prefixed with `self`.)

Local functions and procedures may in turn invoke others, and may be recursive. Standard procedural invocation rules and semantics apply. One-way `local` operations are also allowed. Invocations are interpreted as structured "gotos" in which control does not return to the calling operation. For this reason, procedural operations may not invoke `local` one-ways.

The execution state of objects is in general unbounded. The existence and value of representational bounds for particular classes and objects may be conservatively assessed via static analysis of `local` operations. When bounds are not discerned or discernable, ODL implementations may establish maximal per-object run-time size limits and handle overflow as a run-time error.

Construction

Every instantiable class declaration automatically results in the definition of a corresponding new operation in class System. The new operation has arguments corresponding to all slots declared as <>, and returns a unique link value referencing an object of the indicated class. Implementations of new (as well as delete) are not definable within ODL, although provision of implementation-specific blob-based classes and object layout rules may make them so.

Without qualification, the class's new operation may be invoked anywhere. The visibility of new may be controlled via a generator clause in a class declaration. A generator clause names the classes of entities that may invoke new for the class.

Destruction

ODL message passing rules assume preservation of referential integrity. Objects that may still receive messages may not be deleted. This is best implemented using automatic storage management (garbage collection). However, a delete operation is also associated with each concrete class. Visibility is also controlled via generator.

2.3.2 Constraints

Constraints list properties of instances without otherwise defining concrete forms of slots. Any class may include any combination of constraints and concrete definitions for any slot. A class describing constraints but leaving one or more slots otherwise undefined is termed *abstract*. Abstract classes are not instantiable.

ODL constraints are *partial* descriptions. Objects obey declarative constraints, but any behavior that is not ruled out is possible. The forms of constraints are limited to those that may be evaluated via combinations of static symbolic analysis, translator assisted instrumentation, and dynamic checks. Moreover, translators are only required to perform a subset of these measures, as described below. ODL does not specify whether or how conformance to others is enforced. ODL rules represent a compromise between expressive power and the inferential and run-time requirements upon implementations. Reification of specification constructs renders ODL at best incomplete as a declarative language. However, common constraints remain simply expressible and checkable.

Bindings

In ODL, the bindings from names to all slots except stored functions are fixed and common to all instances of a class. (This restriction may be lifted in a future version.) Bindings for stored slots may be changed during execution (via :=) unless they have been qualified as fixed. The binding for a slot qualified as fixed remains constant across the lifetime of each instance. The qualifier `fixed` may also be applied to a non-stored function to indicate that its value does not vary over time. (Note in this case that `fixed` refers to the value, not the binding.) The keyword `own` is an abbreviation for `local fixed unique`.

Types

All arguments and results (including function values) for all slots must be constrained by type, and optionally by qualifiers `unique`, `common`, and `opt`. Annotations for link types are partial specifications – they list *a* type for the link, not necessarily the *maximal* (most exact) type.

In ODL, a message may be sent only if the recipient will eventually accept it. Determining conformance with this rule is in general undecidable. However, ODL programs must obey the weaker rule that a message be listed in a concrete definition only if it is provable that the recipient does not forever `pend` the message; i.e., if the message triggers a corresponding operation in at least one condition in a class declaration. The proof method is conservative, and based on type checking. In ODL every link is qualified with a type annotation indicating a class to which the referenced object must belong. The ODL type checker treats any attempt to send a message not listed in the indicated class (or superclass) as "not provable", thus as a programming error. However, the checker also admits invocations nested within conditionals checking (via "`in`") that the recipient is of a class supporting an operation. For example:

```
class A op m ... end end;
op calla(a: Any) { if a in A then a.m end; }
```

Function Value Constraints

Additional constraints may be declared via:

- Invariant (`inv`) expressions that hold whenever objects are not engaged in operations (i.e., at all quiescent states).
- Short forms of equality invariants for functional slots: `fn f ...=` *exp*.
- Initial condition (`init`) expressions holding upon construction.
- Short forms of equality-based initial conditions: `fn f ...init=` *exp*.

All constraints and conditions must be expressed within the (executable) value expression sublanguage of ODL. Invariants and initial conditions are boolean-valued expressions constructed from:

- Literals of value types.

- Equality operators (`=`, `~=`) on values of all types.
- Relational ordering operators (`<`, `<=`, `>`, `>=`) on values of `int`, `char`, `real`, `time` types. (ASCII compatible ordering is required of `char`.)
- Operators defined on boolean values: `/\` (*and*), `\/` (*or*), `~` (*not*), `=>` (*implies*), and comma (`,`) (low precedence *and*). *And* and *or* are "short circuiting": successive terms of expressions need not be well-defined if the truth value is determined by previous terms.
- Operators defined on integer values (`+`, `-`, `*`, `div`, `mod`).
- Operators defined on real values (`+`, `-`, `*`, `/`).
- Real-valued mixed mode operators between integers and reals (`+`, `-`, `*`, `/`).
- Operators defined on time values (`+`, `-`), plus mixed mode `*`, `div` operations with integers.
- `if` *exp* `then` *exp* ... `else` *exp* `end`.
- Operator `null(`*link*`)`, defined on link types.
- Operator *link* `in` *class*, that is true if the object referenced by *link* is an instance of *class* or a subclass thereof.
- Field and subscript selection on records and arrays.
- Invocations of functional operations.

This sublanguage may be viewed as a very small pure functional language. ODL restricts the forms of functional operations in order to enable their use in declarative constraints. They thus play a dual role. From a computational perspective, they are restricted forms of operations, but from a declarative perspective they serve as symbolically tractable functions. The requirement that functional expressions be translatable into particular executable forms limits power and restricts expression. For example, bounded universal quantification is expressed via type annotations for function arguments.

In fact, `inv` and `init` are treated by translators as special declaration forms of ordinary functional slots. All `inv` constraints for a class are collected (clausally conjoined in listed order) in executable form as callable `fn inv: bool` that may be invoked at run-time. Similarly for `fn init: bool`. ODL does not otherwise require translators to perform symbolic analysis on constraints (e.g., to determine whether they are even satisfiable). Design checkers that perform such analyses may be constructed, but these capabilities are not demanded of translators.

Operation Constraints

Guards. Guards are "active" preconditions listing the conditions under which nonfunctional, nonlocal operations may be executed. ("Passive" preconditions describing alternatives within accepted operations are listed instead as ifs within effect expressions.) There are both "outer" and "inner" guards, of the form:

```
class C ...
 when c1 then
   op m1
     when m1c1 ==> ...
     elsewhen m1c2 ==> ...
     else ...  end
   op m2 ...
 elsewhen c2 then
   op m1 ...
 else ...  end
end
```

Any combination of "outer" clauses with embedded operations, and "inner" guards nested within operations are permitted. Stylistically, outer guards refer to object state, while inner forms refer to properties of message arguments of ops. Nested sets of guards are also permitted. Different *versions* of the same op may be declared in different arms of outer guards (as seen above for m1. Consistency rules for multiple versions are described in section 2.4.

Boolean expressions within **when** clauses are constructed using the above expression sublanguage. Translators must provide interference-free translation of guard expressions into executable form.

Like **elsifs** in most languages, the condition in each **elsewhen** clause is interpreted to include the negation of all preceeding conditions. This guarantees mutual exclusion of conditions. For example, **elsewhen** c2 above is interpreted as **when** ~c1 /\ c2.

Pending. Pend is a pseudo-effect indicating that a message does not trigger an operation at all under the listed condition. If any other condition ever becomes true, the corresponding operation will be triggered accordingly. All explicit guards and non-local operations are considered to be encased within:

```
class ...
 when ready
   ...
 else
```

```
    pend
  end
end
```

Ready is true when an object is not otherwise engaged in an operation. (This function does not actually exist and cannot be expressed in ODL.)

By default, if an op is listed in only one outer when then it it assumed to pend in all others in which it is not listed. Completely unlisted messages pend forever. Functional operations may pend only when an object is busy in another operation. Local operations are not processed as messages and cannot pend. Thus, no explict guards may be associated with functional and local operations.

Translators may employ any non-interfering mechanism to implement pend. The use of guards does not require that translators establish identifiable per-object message queues. No ODL constructs refer to queues. No run-time support is needed if a translator can determine that no pend conditions can ever be encountered for an object. When any of several messages may be accepted (i.e., clear guard conditions), any one of them may be chosen. Implementations may provide stronger ordering guarantees. Implementation limits in the number of possible simultaneously pending messages are permitted.

Translators cannot always statically detect situations in which conditionally accepted messages forever pend. They may provide run-time mechanisms to assist users in dealing with resulting deadlocks and overflows.

Effects. Effects (==>) list conditions that must hold as a result of particular operations. Clauses within effect descriptions are defined using the constraint sublanguage extended with constructs:

exp' The value of an expression as evaluated upon completion of the operation. (Unprimed forms within effects refer to evaluation upon commencement of operations.)

msg' Assertion that the effects of msg hold at completion of the operation.

msg'' Assertion that the effects of msg hold (perhaps only briefly) at some point after commencement of the operation.

@boolexp The time after which the operation commences at which the expression becomes true.

No translation into executable form is specified for effect expressions. In particular, effect expressions with double-primes cannot be translated in any useful manner since the time at which they should hold true is unbounded. However, static analysis tools may be constructed to determine satisfiablity of effect expressions and partial conformance of concrete forms. Other tools may be devised to help instrument checks for certain effects and/or to help generate test code.

ODL effects are *not* subject to "frame assumptions" that claim that properties that are not mentioned do not change. Any behavior consistent with constraints is allowed. All properties that must be preserved within ops should be explicit unless they are also listed in invs (which serve as implicit pre- and postconditions for all operations). In contrast, functional operations do obey frame axioms since they are computed in a side-effect-free manner.

2.4 Inheritance

A class may list any number of superclasses. If none are listed, the class is taken to be a subclass of Any. Any is the root of the class system. It defines only the fixed slot `self`, providing a reference to self. Subclass declarations extend the scopes of their superclasses. The declared class structure determines the type structure for links. The type of a link referencing instances of a class is a subtype of superclass link types.

Subclass declarations extend those of their superclasses. All stated properties (slot definitions and constraints) in superclasses are preserved in subclasses. Additions may not invalidate superclass properties. Detected conflicts may be trapped as programming errors by a translator, but it is not specified whether or how conformance is enforced.

Subclass declarations take two forms, adding new slots and adding (strengthening) features to those listed in superclasses. Additions are conjoined to the corresponding superclass declarations. Full redeclarations are unnecessary except in those cases where features cannot otherwise be expressed (e.g., when adding qualifiers). When a new slot has the same name as one in the superclass, or when two or more superclasses have slots of the same name, the following rules apply:

Multiple Declaration. Two declarations with the same slot type (operation, procedure, function, local), number of arguments, argument types and qualifiers denote the same slot. All other aspects of the declarations are coalesced as one.

Versioning. Two non-local, non-functional declarations differing in the declared *link* type of one or more argument fields, or differing in outer when guards are considered to be variant *versions* of the same slot.

Overloading. Two declarations differing in number of arguments, or in the types of one or more argument in the case where those types are incommensurate (e.g., `int` versus `real`, any non-link type versus a link type), or where only one is declared as `local`, are taken to be two unrelated slots that may coexist (i.e., as a case of *ad hoc* overloading). Variants with and without opt qualifiers on one or more arguments are similarly treated as overloaded.

Unsupported. Two declarations differing in any other way (e.g., `fn` versus `op`, qualifiers) are disallowed because invocation forms of the different cases cannot be distinquished in ODL.

The first two cases are instances of adding features to existing slots. The following additions are permitted. Analogous rules apply when merging the declarations of two or more superclasses.

- Adding a concrete definition to an existing slot.
- Adding a new clause to an existing effect.
- Adding a nonconflicting `inv` constraint. All `inv` clauses in the class and superclasses are interpreted as a single conjoined expression (with subclass clauses prepended to superclass clauses) that must be satisfiable.
- Adding a nonconflicting `init` constraint.
- Adding a constraint to a functional slot. The type of a function may be strengthened by replacing the result type declared in the superclass with a subtype thereof, adding `common`, `unique` and/or `fixed`, or removing `opt`. However, these may not conflict with other superclass constraints or definitions. For example, it is illegal to add a `fixed` qualifier if a `fn` value varies within a superclass operation.
- Adding a constraint to a procedure result. Result types and qualifiers of anonymous replies may be added in the same manner as for functions. Also, a constraint that one or more alternate named replies are not issued may be indicated by omitting them in the redeclaration. (The converse case of adding alternate named replies is not allowed.)
- Adding or subdividing an outer `when` clause into two or more subconditions in order to add a new operation or another version of an operation under one or more of them.
- Adding or subdividing an inner `when` clause into two or more subconditions in order to add a new clause to an effect and/or add a concrete definition under one or more of them. Effects and result types of all versions must meet all applicable superclass constraints.

These rules apply whenever a subclass adds a new version of a slot or two superclass versions exist. All declarations of different versions are interpreted as if they were different arms of a single operation with multiple `when` guards. A translator may fabricate guards (and/or perform equivalent transformations) in any way consistent with the declarations. For example, in:

```
class A ... end
class B is A ... end
class C ... end;
class D
  op m(a: A) ==> ea end;
  op m(b: B) ==> eb end;
  op m(c: C) ==> ec end;
end
```

The declarations of m in D may be transformed into a single operation, with all argument types recast in terms of their nearest common superclasses (here, just Any):

```
  op m(x: Any)
    when      x in B ==> eb
    elsewhen  x in A ==> ea
    elsewhen  x in C ==> ec
    else pend  end
```

The implicit negation of preceding guards in when clauses transforms consistency issues to ordering issues in the equivalent ODL code. For example, this may also be transformed as:

```
  op m(x: Any)
    when      x in C ==> ec
    elsewhen  x in B ==> eb
    elsewhen  x in A ==> ea
    else pend  end
```

In both cases, the version for B specializes that for A. However, class C bears no subclass or superclass relation to the others, so the clause may be considered in either fashion, at the discretion of the translator. For example, if the operation were invoked with an argument of type AC (a subclass of both A and C) then either version might trigger. While not disallowed, such non-determinism should be avoided.

The results of different versions of procedural operations may also vary. The combination rules are the same as would apply if each reply were considered as an operation proper. The base version is constructed by conjoining all cases as multiple replies, while merging (as the nearest common superclass) the types of identically named (or nameless) replies in those cases where fields differ only in link type. For example:

```
class E
  op p(a: A) f: A ;
  op p(b: B) g: B ;
  op p(c: C) h: int;
  op p(d: D)  ok(), bad(i: int)
end
```

This may be represented in the form:

```
class E
  op p(a: Any) fg: A, h: int, ok(), bad(i: int);
end
```

When a superclass version of the procedure exists, the resulting return expression must conform. In particular, added named reply forms resulting from such combinations are not allowed.

2.4.1 Subclass Restrictions

A constraint of the form `C = oneOf(S1, S2, ... Sn)` limits the declarable subclasses of `C` to those listed under `OneOf`. Stylistically, `OneOf` is used to indicate that the listed subclasses are the only ones logically possible. For example, a class with a `fixed bool` slot might be partitioned into two subclasses with it set to true versus false. Also, subclasses defined via `OneOf` serve as analogs of enumeration types found in other languages. A translator may enforce and exploit the facts that partitioning constraints place fixed bounds on the number of subclasses and/or that partitioned siblings may never have a common subclass.

2.4.2 Defeasible Inheritance

A class may list another "base" class in an `opens` clause to indicate that it shares all but specifically redeclared features with this base. The rules are the same as those under normal inheritance except that *any* feature may be redeclared in *any* way – declarative constraints in the base declaration are ignored. Additionally, any unguarded base slot may be redeclared as `local`. The class listed in `opens` is not a superclass; link types of the derived class are not subtypes of those for the base.

2.5 Parameterization

Classes, operations, and records may be paramaterized with one or more class arguments in brackets appended to the declared name. They may may then appear anywhere an ordinary type could appear inside the declaration. A parameterized entity simultaneously defines all possible specializations of that entity. The specializations are themselves ordinary classes, operations, and records, subject to normal use. Parameterized classes may be defined as subclasses of other parameterized classes.

Translators must generate specializations for all versions that are actually used in a program. They may generate others as well; for example, those used in possible but untaken computation paths. Because they remain controversial, ODL does not support *kinds* (types of types). There are no type constraints on type arguments. However, translators must detect and report errors when expansions result in specializations that contain (nonparameterized) type errors. Since instantiation is static, first-order type rules are maintained for all specializations used in a program.

A complete EBNF syntax for ODL can be found in Appendix A. The partially predefined class `System` and single instance `system` are handled differently than all others. There need not be a single identifiable `system` object during execution. Its functionality may be spread (perhaps redundantly) across all processes or otherwise achieved in a constructed system.

Chapter 3

Compiling ODL in DIAMONDS

The goal of the Diamonds system is to efficiently support a fine-grained object oriented language in a sound software engineering manner. Towards this end, Diamonds itself has been designed and developed in an object oriented approach. The system can be viewed as a set of classes that are organized into libraries/directories which may be used to compose the tools of the system.

In this chapter the design of the language and compilation components of Diamondswill be examined. This will include a look at the major subsystems within Diamonds and the Classes developed. Appendix B discusses the actual organization of the tools with C++, and the tasks involved in building the components.

The Diamonds software system components can be conceptually broken up into two primary areas of concern: language processing, and runtime support. Because of the re-use of some classes both in-toto and as base classes which are further refined, there is a fair amount of overlap of use from the language processing tools to the runtime. In the sections that follow we discuss each category of classes as if they were a separate entity. The reader should keep in mind that the final tools are a blend of the classes from each category.

Our discussion of the classes will roughly follow the flow of a typical odl program through the system. This begins with its parsing and analysis and ends with its execution in the distributed runtime environment.

3.1 Language Processing Components

At the front end of our compiler we have made use of the typical Unix based language tools. This includes the lexical analyzer generator, lex [LS86], and the LALR parser generator, Yacc [Joh86], and tools compatible with it such as GNU's flex and Bison. odl is a relatively small and simple language when compared with something like C++. To illustrate this we might use the number of rules in their LALR grammars as a rough metric. A publicly available C++ grammar[1] contains approximately 168 rules, most of which contain many complicated productions. Our odl grammar has about 85 rules, most of which have relatively few and simple productions.

3.1.1 Parser Representations

Most traditional compilers use a symbol table to store information about language constructs as they are encountered during parsing. We have taken a different approach to the need to store this information. The program itself is represented as a set of *classes* that are responsible for the maintenance of all information about themselves. This includes attributes such as,

- The name of the class
- The methods supported by this class. This is ultimately the storage place for the actual code for this method too.
- The number and type of static, data attributes
- The classes this class might be inheriting from
- etc.

These internal representations of the the actual odl classes become the keys for traditional parsing tasks such as scope resolution. A class maintains information about the scope in which it was declared. This information can then be used later to look up variables as they are used. One way of viewing our internal representation of odl is as a *meta-class* type arrangement. The parser *meta-classes* describe the characteristics of the actual odl classes as well as the contextual environment in which they were declared.

To complement this object-oriented approach to parsing, we have also reified many of the abstract constructs within the grammar. The effect of this is that the productions in the grammar become classes in the parser representation. Some of these classes are transient and carry information for only a few rules up the parse tree while others live on and become final components of the parsed representation.

As a simple illustration of this design technique, consider the fragment from the grammar in figure 1. In this figure the grammar for a `reply` is given. In odl, a `reply` can have an expression as an optional argument. This is all captured by our `Reply` class that stores this information. A `Reply` itself is a particular type of `Statement`, which in turn itself inherits other properties.

[1] Version 2.0 of Jim Roskinds grammar [Ros91]

```
Reply:
          REPLY          { $$ = new Reply(0);  }
        | REPLY Exp      { $$ = new Reply($2); };
```

Figure 1: Grammar fragment – Reply

3.1.2 Abstract Syntax Tree

In the design of the parser, we attempted to distinguish between those classes that lend a support role, and those which represent a particular aspect of odl. The majority of odl is message sends. These message sends are just a particular type of an expression within odl. To complement our *symbol table-less* design discussed in the previous section, we also use a tree type representation for statements and expressions. The leaves on this tree are actual odl expressions, which include simple operations like addition, subtraction etc.

This design has implications on the code generation phase of the compiler. In this case we are able to localize all information about code generation within the class for each expression type. This allows a code generation protocol to be devised that is basically,

```
foreach element in the expression list do
   element -> generate code for yourself
end
```

If the particular element contains sub-expressions, this code-generation request is forwarded on to them.

3.2 Intermediate Representation

Our goal of supporting odl within Diamonds across a heterogeneous set of machines almost precludes the option of generating code, apriori, for a specific architecture. Instead we have designed an intermediate representation that captures the semantics of the odl program which is being parsed in a machine independent manner. We refer to this intermediate format as SIRF [2].

One of the other challenges in the design of SIRF is the goal of allowing compilation of SIRF code on the fly, as the program is being interpreted. This requirement tends to favor a format for SIRF code that is as close to the machine level code as possible. We have chosen to design

[2] SIRF is an acronym for Syracuse Intermediate Representational Form

SIRF as a register oriented language with as few instructions as possible. In this respect it bears a strong resemblance to many of the RISC-based processor instruction sets such as the Sun SPARC [Inc87] or IBM POWER [Cor90] that are common on workstations today. SIRF currently has 29 instructions which can be divided into categories as shown in table 1.

Type	Number
Arithmetic	8 instructions
Relational	6 instructions
Jumps	3 instructions
LoadStores	4 instructions
Messages	6 instructions
Misc	2 instructions

Table 1: SIRF Instruction Categories

Tables 2 and 3 descibe the SIRF instructions in more detail. Table 2 shows a mnemonic and an integer descriptor for each instruction, along with the number and general semantic meaning of each argument. In the table, any phrase ending in `Reg` refers to a register in the `VirtualMachine` and those begining with `op` denote operands. A formalized description of the semantics of each SIRF instruction is shown in table 3. Here, the results of the instructions operation are specified using a "C" like syntax. For instruction like `Load` or `Store` where register variables are treated as pointers, they are shown de-referenced via the C unary * operator.

The actual number of instructions could be reduced if needed. There are a few complementary instructions such as `JumpTrue` and `JumpFalse` that could be replaced by a single instruction at the cost of some additional generated code. As a design decision though, the approach was taken to include both types of these instruction so as not to complicate or preclude local machine optimizations when actual native code is generated.

3.2.1 Creation

From a class perspective, an odl program is nothing more than a nested set of classes that begins with the predefined, outermost class, `System`. Our final SIRF representation of odl is no different.

Each odl *meta*-class is responsible for the creation of a `SirfClass` that corresponds to the odl class. The main difference between the odl and SIRF classes that in the place of the abstract syntax tree representation of `Statement` and `Expressions` are SIRF code substitutes.

SIRF code is generated as the final step in the compilation process. Starting from the root of the class implementation hierarchy, `System` is asked to generate a `SirfClass` representation

Instruction Mnemonic	Integer Id	Arg1	Arg2	Arg3
NoOp	1			
Add	3	op1Reg	op2Reg	resultReg
Subtract	4	op1Reg	op2Reg	resultReg
Multiply	5	op1Reg	op2Reg	resultReg
Divide	6	op1Reg	op2Reg	resultReg
xDIV	7	op1Reg	op2Reg	resultReg
xMOD	8	op1Reg	op2Reg	resultReg
Negate	10	opReg	resultReg	
Not	11	opReg	resultReg	
Equals	13	op1Reg	op2Reg	resultReg
LessThan	14	op1Reg	op2Reg	resultReg
GreaterThan	15	op1Reg	op2Reg	resultReg
NotEquals	16	op1Reg	op2Reg	resultReg
GreaterEquals	17	op1Reg	op2Reg	resultReg
LessEquals	18	op1Reg	op2Reg	resultReg
Jump	20	target		
JumpTrue	21	target	condReg	
JumpFalse	22	target	condReg	
LoadLit	24	value	destReg	
LoadSelfLink	25	destReg		
Load	26	baseReg	offset	destReg
Store	27	baseReg	offset	sourceReg
Move	28	sourceReg	destReg	
MsgCreate	30	size	methodId	destReg
MsgDispatch	31	syncFlag	msgReg	
LiteNew	33	classId	destReg	
HeavyNew	34	classId	destReg	
DollarDispatch	35	synchFlag	msgReg	
Return	36	resultReg		

Table 2: SIRF Instruction Labels and Arguments

Instruction Mnemonic	Semantic Meaning
NoOp	no operation
Add	resultReg := op1Reg + op2Reg
Subtract	resultReg := op1Reg - op2Reg
Multiply	resultReg := op1Reg * op2Reg
Divide	resultReg := op1Reg / op2Reg
xDIV	resultReg := (int)op1Reg / (int)op2Reg
xMOD	resultReg := op1Reg % op2Reg
Negate	resultReg := -opReg
Not	resultReg := (opReg > 0) ? 0 : 1
Equals	resultReg := (op1Reg == op2Reg) ? 0 : 1
LessThan	resultReg := (op1Reg < op2Reg) ? 0 : 1
GreaterThan	resultReg := (op1Reg > op2Reg) ? 0 : 1
NotEquals	resultReg := (op1Reg != op2Reg) ? 0 : 1
GreaterEquals	resultReg := (op1Reg >= op2Reg) ? 0 : 1
LessEquals	resultReg := (op1Reg <= op2Reg) ? 0 : 1
Jump	pc := target
JumpTrue	pc := (condReg > 0) ? target : pc
JumpFalse	pc := (condReg < 0) ? target : pc
LoadLit	destReg := value
LoadSelfLink	destReg := objPtr
Load	destReg := *(baseReg + offset)
Store	*(baseReg + offset) := sourceReg
Move	destReg := sourceReg
MsgCreate	destReg := msgPtr
MsgDispatch	*(msgReg) message is sent
LiteNew	destReg := objPtr
HeavyNew	destReg := objRefPtr
DollarDispatch	*(msgReg) message is sent
Return	virtualMachine.resultReg := resultReg

Table 3: SIRF Instruction Semantics

of itself. This in turn will cause additional SirfClasses to be generated as the classes within System are asked to generate SIRF images of themselves.

As part of this generation process, the result of every Expression element is assigned to a register. The register allocation algorithm used in the prototype implementation is the simplest one possible. An infinite set of registers is assumed and a new register is assigned to the result of every expression. Register allocation is reset at method boundaries. So every method begins with assignments to register 0 and then it goes up from there.

The result of the SIRF code generation phase is a parallel set of classes to those which were created when the program was parsed. The difference is that this set is now specified in terms of our intermediate representation and is targeted towards our VirtualMachine. The details of the design of our VirtualMachine will be covered in section 3.3.1.

The final step in this process is to preserve the SIRF representation in persistent storage by having it write itself out to a file. For this task, each SIRF instruction knows how to write itself out. To allow the individual instructions to be distinguished at a later point, each type of instruction is assigned a unique number. This is similar to the "bytecode" representation used in Smalltalk [Gol83] or Self [CUL89]. In the prototype version of SIRF code, however, the instructions are not squeezed down into a "bytecode" format. This is primarily for debugging, testing and evaluation purposes. Perhaps a better description of SIRF code at this point would be that it is in a "wordcode" format rather than a "bytecode" format. Nothing precludes the generation of true "bytecodes" at a later time.

3.2.2 Optimizations

There are several optimizations that could be performed both while SIRF code is being generated and after. The first would be to implement a simple but more sophisticated register allocation algorithm. This could save on some of the memory management overhead during interpretation. Note that since we are still dealing with an abstract machine, another register allocation phase would have to be run if and/or when native code is generated.

All of the traditional code optimization techniques can also be applied at the SIRF code level. This will pay benefits across all architectures because techniques such as common subexpression elimination or dead code removal are not hardware specific. More local optimization such as the use of machine idioms will have to be deferred until native code generation.

3.2.3 Reification

For the interpretation or execution of SIRF, the preserved file image must be read and the set of SirfClasses and their representations instantiated. This is a straightforward process where the file image data is used to determine the type of SirfClass or SIRF instruction to instantiate.

Just as in the process of writing out the SIRF code representation, since we localized this responsibility within each and every instruction, the process of reading in a new instruction is the same. Each instruction knows how many arguments to pull off the input stream and what types of arguments to expect. This type of localization makes any changes in format for an individual SIRF code instruction visible only to the instruction itself. No other implementation code need be modified.

Once instantiated, the `SirfClass` structure within the interpreter or within the runtime entities (as discussed in the following sections) is *identical* to that which existed within the parser. This fact is used to allow the same interpretation engine to be used as both an option at the end of parsing, or as a later stand alone operation.

3.3 Basic Runtime Infrastructure

Our runtime infrastructure starts with a set of semi-abstract components that define the common protocols and data for other layers. The simplest and most basic component is the `VirtualMachine` that defines the substrate upon which SIRF code executes in our interpretive environment. The design of the `VirtualMachine` and the set of instructions which operate on it (SIRF) actually goes hand in hand. In our case, since we will be targeting SIRF towards more than just our `VirtualMachine`, we stressed the design of SIRF over that of the `VirtualMachine`. The requirements that SIRF be retargetable to a wide variety of machines led to the design of `VirtualMachine` and other infrastructure components that are relatively clean and simple.

3.3.1 VirtualMachine

The `VirtualMachine` contains a set of registers, a program counter and a status indicator. For prototyping purposes the set of registers are resizable, with an upper bound limited only by the actual amount of memory available to the program. In interpreting or scheduling an object oriented language like odl, there is not really a need for a vast number of registers. Most methods are small, a few lines at most, and in turn have only a few locals and temporaries. This behavior has also been observed for C++ programs [CGZ94].

While a program is executing, the program counter of the `VirtualMachine` points to the next line of SIRF code to be interpreted. The status indicator signals the current state of the `VirtualMachine` as either running or context switching.

The `VirtualMachine` has four registers which are reserved. These registers and their semantics are described in table 4. [3]

[3] The `ObjDataReg` register is needed in the prototype because, in the C++ layout of objects, the base address of the allocated object and the addresses of its addressable `RegisterBlock` are not the same. In implementations that do not use `RegisterBlocks`, this register will not be needed.

Register	Use
ContextReg	Contains a pointer to the currently active Context
ObjAddrReg	Points the base address of the currently active Object
ObjDataReg	Points to the first addressable word within the currently active Object
ResultReg	The result of a previous function call is stored and accessed via this register.

Table 4: VirtualMachine Reserved Registers

Since this is a virtual machine, intended for interpretive use, it does not contain registers or bit flags for conditional evaluations. Instead, these sorts of flags are sensed and acted upon directly by the SIRF instruction which is executing. For example, consider the instruction JumpTrue which has two arguments: a register to test and an instruction number to set the program counter to if the test is positive. In the course of executing this instruction, the value contained in the register will be examined. If it is found to be greater than zero, then the program counter is set to the jump target instruction. No other VirtualMachine registers or flags are used or set during this execution step.

3.3.2 ExecutionAgent

An ExecutionAgent is our unit of local concurrency. In a pure odl implementation or within a single processor machine in Diamonds, each VirtualMachine and Supervisor (see section 3.3.4) will have a single ExecutionAgent. The ExecutionAgent is an intermediary between the VirtualMachine and the Supervisor. It is responsible for initializing the Context(see 3.3.3) and VirtualMachine after a context switch or rescheduling. It also performs the instruction fetch and scheduling.

We can handle either true or pseudo local concurrency by associating multiple (ExecutionAgent, VirtualMachine) pairs with a single Supervisor. This can also be done by associating several Supervisors with a single address space. This sort of arrangement will be discussed later when we address the uses of Ensembles and Clusters.

3.3.3 Contexts

In a purely procedural language such as C, all function invocations are performed on the "Stack". At the point of a call a new stack frame is allocated. This stack frame (depending on the actual operating system and machine) is large enough to hold the arguments of the function being called and any locals or temporaries that might need to be saved during the execution of the function. When the function is called this new stack frame is used. When the function completes the stack frame can be deleted and control returns to the "previous" stack frame.

In supporting a language such as odl where many calls are asynchronous, we can no longer make exclusive use of a stack frame type model. Instead, we need to allow for an arbitrary calling and callback scheme, where the function that is executed next depends on both who is ready and the scheduling algorithm, rather than which method is on the top of the stack.

We are able to support this arbitrary calling and callback scheme with what we call `Contexts`. A `Context` holds all of the information necessary to invoke and carry out an operation on an object. This includes information about the object being called, the method being called, storage space for local variables and temporaries, and finally, information about *who* is to be called back in the case where the method is synchronous.

The `Context` also stores information about the state of the `VirtualMachine` across context switches. This includes registers which are still "alive" and the current program counter [4].

3.3.4 Supervisor

At the heart of the runtime protocol is a `Supervisor`. The `Supervisor` contains those structures which are needed by any system which will be interpreting, simulating or executing SIRF code. These components include a queue of `Contexts`, at least one `ExecutionAgent`, as well as a handle to the SIRF code for the program that is being run.

The `Supervisor` also handles upcalls [5] from the `Virtual Machine` or `ExecutionAgent`. These calls are to deal with SIRF instructions whose exact semantics vary from environment to environment. For instance, during the process of creating a new object, there is an upcall to the supervisor (or subclass of `Supervisor`) who then handles the details of the call. In the case of a pure odl interpretation, this would result in the allocation of a new object from the heap on the current machine and in the current address space. In the case of the full distributed Diamonds version, this new object could be created on any machine in the network, based on both static hints and dynamic resource constraints.

[4] In the prototype, we avoid the overhead of register spilling and refilling, by allowing the `VirtualMachine` to directly manipulate the register storage area of a `Context`. The alternative would be to copy live data to and from the `VirtualMachine` registers and the `Context` registers at context switches. This is one case where our "virtual" environment allows us to take some simple shortcuts.

[5] For the most part, upcalls in our environment correspond to virtual function dispatches in C++

As method invocations are made during the course of execution, new `Contexts` are created and queued in the `ContextQueue` of the appropriate `Supervisor`.

3.3.5 Other Components

There are several other components that are central to the runtime infrastructure. These deal for the most part with implementing the behavior as described by the odl program.

Objects

At runtime, we must allocate space to store state information associated with odl objects which have been created. We do this quite naturally with our own abstraction of an object, `class Object`. The `Object` has enough space to store all of the stored attributes of its corresponding odl class. In the prototype implementation we do this with `RegisterBlocks` and `StoredAttributes` which are discussed in the following sections labeled **Registers** and **Attributes** respectively.

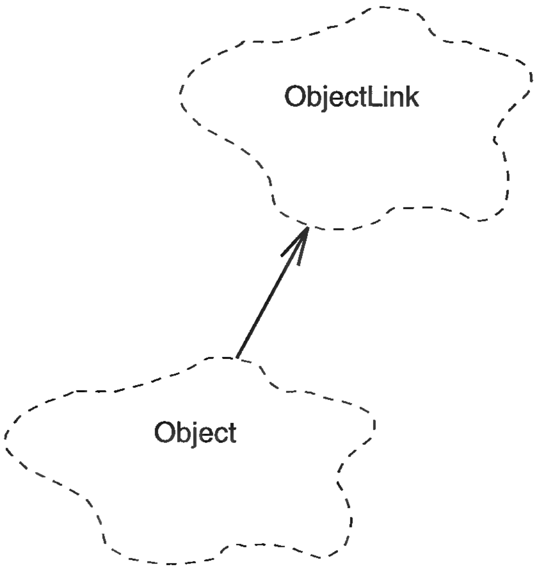

Figure 2: Object and ObjectLink Class Diagram

An `Object` is an instance of an `ObjectLink` class as pictured in figure 2. The relationship between an `Object`, an `ObjectRef` [6], and an `ObjectLink` allows the runtime system to manipulate Objects and pointers to objects (Links) as if they were equal, in many cases.

[6] See section 5.4 for more information on references and related types of references like `ObjectRef`

Attributes

An object can be thought of as a container which encapsulates data. The data which may be encapsulated within odl is of the following basic forms,

1. A *computed* attribute. This is an attribute that is determined by a method or function call.

2. A *stored* attribute. This is an attribute of a primary, odl type (such as `int`, or `char`, etc.) that is assigned space within the object for storage.

3. A *link* attribute. This is an attribute that names another object. (This is actually a type of stored attribute).

To track types at runtime for coherence and debugging abilities, we use a `StoredAtt` class. A runtime `Object` can then be thought of as a set of slots, each one being an attribute of some kind. Each slot may store any value that is in the union of all odl builtin types. The `StoredAtt` class is designed to maintain information about the type of value that is stored within its slot and raise an exception if it is accessed illegally(eg. an `int` value being accessed as a `char`).

Registers

In mapping our attributes, as discussed in the previous section, to a physical machine, the slots that we mentioned would become addressable chunks of memory that are moved back and forth to local CPU registers for manipulation.

We keep a similar model in the runtime, with the expectation that the mapping of the runtime to a physical machine will be easier because of it. To do this, we equate our `StoredAtt` class with registers in the `VirtualMachine`. So, a virtual machine register is a `StoredAtt`, and thus knows the type of value which was last written to it.

This uniform view of attributes and registers allows us to make another simplification in the runtime. An `Object`'s "data space" can now be modeled as a set of "storedAttributes" which in turn are synonymous with registers. We formalize this with an actual object, a `RegisterBlock`. A `RegisterBlock` is a resizable array of `StoredAtts`. This same component is used with the `VirtualMachine` to store data, as well as within a `Context` to store local and temporary variables.

There is some overhead cost paid for this approach in the form of extra memory allocation and additional calls for type checking and use. We feel this cost has been more than balanced by the benefits received in terms of simplified error checking and a cleaner design.

Messages

The only means of communicating among objects within odl is via messages. A message is sent to access an attribute. If the attribute is computed, then a `Context` will be built and executed. If the attribute is stored, then it can, perhaps [7], be directly accessed.

The information within a message is captured by an `ODLMessage`, as pictured in figure 3. An `ODLMessage` contains a header which describes the total size of the message (in slots) and information about the target object and method. The remainder of the message is the body, which is a `RegisterBlock` as described in the section labeled **Attributes**.

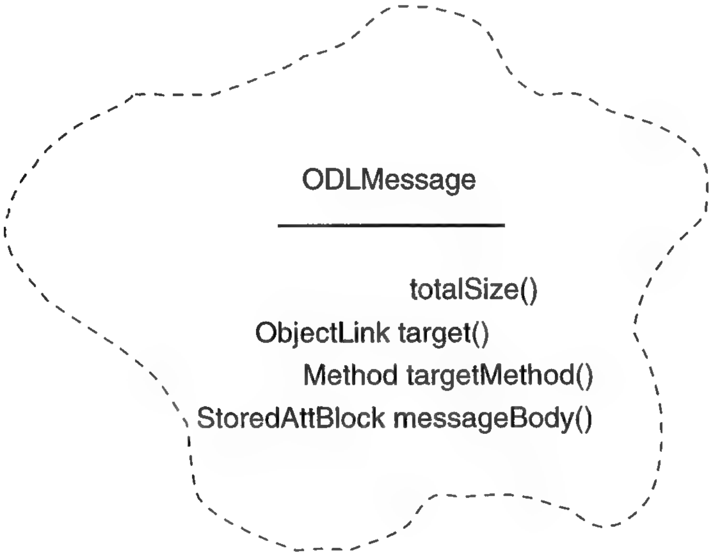

Figure 3: ODLMessage Class

An `ODLMessage` in the format as described in figure 3 only has meaning within a single address space, since it makes use of pointers to refer to the target object and method. If a message is to be sent to an object in the distributed version of the runtime, additional naming conventions will be needed to replace the pointers. This is described in section 5.3.

[7] Remember, in the distributed case, remote objects will not be directly accessible.

3.4 Simulation

The Simulator is a specialized version of a Supervisor. It is our pure odl interpretation environment. By this, we mean that there are no clusters, ensembles, or distributed execution within the Simulator.

The Simulator is a component of two Diamonds subsystems. It is integrated into the parser as a back end option. This allows an odl program to be interpreted directly without the extra steps of writing out SIRF code and the invocation of a separate interpreter.

The Simulator is also at the heart of our stand-alone interpreter.

3.5 Summary

In this chapterthe design of an object oriented infrastructure for language processing has been presented. Prototype tools for the parsing of odl and its interpretation have been designed and built. As part of this work, we have also designed an intermediate representation for odl, SIRF, that is machine independent and amenable to various optimization techniques which are presented in part in [Sha94].

This work is unique, and makes the following contributions,

- a model for constructing object oriented language processing tools
- an novel internal representation for a language that does not make use of a symbol table. Our approach instead models language components as objects and delegates many of the traditional compilation tasks such as code generation to them.
- the design and specification of an intermediate language for the interpretation and compilation of fine-grained active object languages.

Chapter 4

Object Clustering

In the previous chapters we have concentrated on fine-grained active objects, where the phrase "fine-grained" refers to the requirement that every object obey these properties, regardless of its size or complexity. Although object-models of this variety are very powerful they create severe problems to the run-time system. If represented in a literal fashion, each active object would require its own process.

In trying to support such a programming model in a distributed setting we realize that it is not efficient to maintain this level of granularity at the operating system level. The overhead caused by managing these fine-grained entities overwhelms the advantages gained. The additional complexity introduced by distribution and it's related issues such as migration and heterogeneity force us to seek another, coarser grained, model to be supported at the level of the run-time system and a method for mapping between these models.

The basis for our approach is the observation that many aggregate abstractions in the design are inherently single-threaded in nature. Our goal is to map all the fine-grained objects into a much smaller set of aggregates each of which have this "approximately single-threaded behavior". The resulting model is one of *clustered* entities that are composed of fine-grained objects. Clusters are units for scheduling, distribution, migration and all other run-time activities. They encapsulate locality of reference inherent in the application and are better suited toward mapping onto a process. The run-time view of a cluster is shown in Figure 4.

Clustering can be defined as the process of mapping the programming model into a cluster-based model supported by the run-time system. Alternatively it can also be defined as a process of informed and intelligent object composition. Clustering brings together objects that are related to one another in a purely computational sense. Refinements of this relation form the basis for cluster formation. Clustering as a means of composition in an object based system

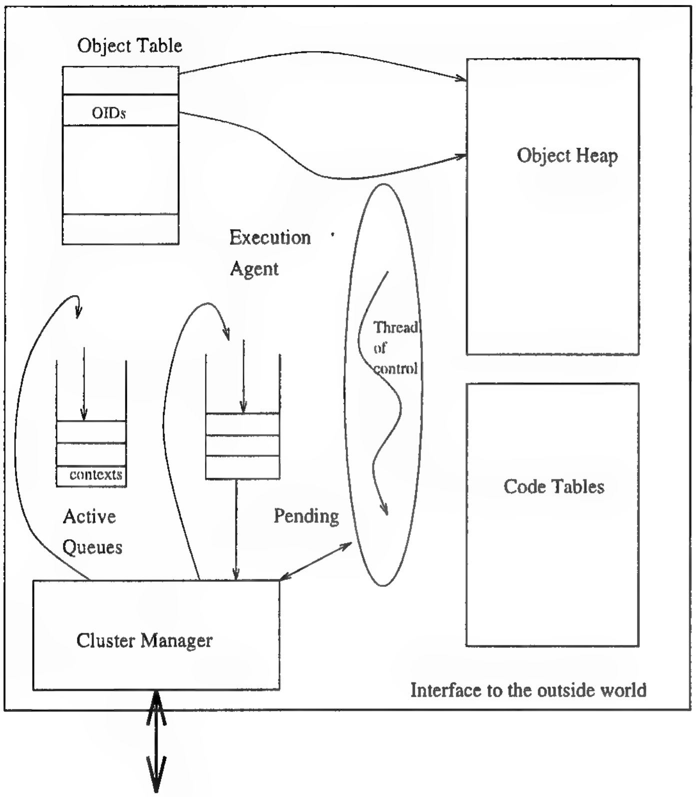

Figure 4: A Cluster in DIAMONDS.

serves to:

Optimize Communication: Late binding of different communication mechanisms so as to use lightweight communication for intra-cluster objects serves to significantly improve the performance of an application.

Structure Concurrency: Objects which perform their activities in parallel can be contralocated (placed on different clusters) while at the same time colocating objects which are as a group are "single-threaded" in nature thereby saving resources.

Improve VM Performance: Clusters encapsulate referential, object locality. This property can be exploited by a virtual memory subsystem which arranges clusters on contiguous pages. Sets of contiguous pages can then be paged in on request, OO design is intrinsically compositional in nature. Clustering, therefore, is a natural process which may occur as part of the design process or be partially automated via clustering tools.

Clustering in object systems is not a novel idea. It has been proposed in several contexts for many reasons. In [Sta84], Stamos advocates static grouping of fine-grained objects in Smalltalk to improve virtual memory performance. Efficient run-time management of objects was the goal of clustering objects in [HK87]. This effort takes into account two dimensions of clustering namely virtual memory performance and improved communication afforded by locality and colocation of related objects. It employs a two level clustering scheme in which objects in a cluster are grouped into regions which represent a contiguous set of virtual memory (VM) pages. In this effort, the user is responsible for creating objects in specific clusters.

Object clusters, termed "clumps" in [Dic91], are used for effective load balancing by retaining locality during object migration. Cluster formation is addressed in Chatterjee's work on clustering objects for efficiency, [CR92]. He proposes a heuristic model of cluster formation where class characteristics are collected via execution profiles and class groupings are arrived at by suitable analysis of the profile information. Dynamic clustering of objects to improve (communication) performance has been suggested in [CBHS93], where the grouping system is integrated with the garbage collection mechanism.

4.1 Clustering Approaches

Clustering of objects, be it for databases, resource management or improvement of VM performance can be done in many ways. One way of categorizing these methods is based on when it is done. There are three possible approaches to the problem:

Static Clustering: Also known as language based clustering, this involves static analysis of the code/object base to determine the clusters.

Dynamic Clustering: Instead of accomplishing partitioning via static analysis, it is possible to do so dynamically at run-time depending on the applications behaviour. An obvious disadvantage is the performance of both the clustering system and that of the application itself since both instrumentation as well as dynamic partitioning are time-intensive operations.

Clustering Via Profiling: A middle ground between the first two categories, this method involves profiling several runs of the application and using the collected data to extrapolate an applications behavoir during subsequent runs. Although it is static in nature, it relies on profiled data collected dynamically. Again the disadvantages of instrumentation costs come into play.

There are also hybrid approaches used such as the one adopted by Concert [CKP93]. One of the major goals of this effort is to demonstrate that a pseudo-static clustering method can accomplish the same goals as that of a profiled method and for lower costs. Note that purely dynamic clustering is infeasible because of the high costs associated with instrumentation and dynamic object mobility necessary. Profiled methods fail in situations where dynamic application behavior is heavily dependent on it's inputs since each run can yield different perspectives of the same program. Static clustering can do no worse in these situations and can be accomplished without the need for costly instrumentation.

4.1.1 Evaluating Designs

Regardless of the methodology used, each method must quantify certain application characterstics in order to effectively partition the application. These characteristics are primarily concerned with estimating the interaction between objects (communication) and the concurrency between them. The metrics can be described in two categories:

Object Metrics: This category of metrics are dependent on the class specification and are constant for any object of the class. They describe the generic characteristics of classes and include:

- *Size:*
- *Computational Complexity:*

Inter-object Metrics: are those which describe the dynamic behaviour of the application in terms of the interaction between objects of different/same classes.

- *Type of Communication:* Whether synchronous or asynchronous.
- *Strength of Communication:* in terms of the bytes of data passed between objects in a message and the wait time of the sender for that message.

4.2 Cluster Semantics

There are many issues involved in clustering objects. A discussion of these issues follows.

Object Model: The basis of clustering is the computational and interaction models that form the core of the programming paradigm. Any clustering methodology therefore needs to take into account the mechanisms available to the developer for communication, concurrency control and the like. Some of the object model features which affect clustering are:

- *Active Vs Passive objects:* In the case of passive object models a key design task, (*physical design*), involves manually describing active agents (processes). Thus, clustering for the purpose of mapping to processors becomes moot. Clustering instead would be most interested in managing locality. In the case of an active object model, one starts from a point where each object is logically assigned to it's own process and clustering becomes a compositional activity.
- *Interaction semantics:* Clearly the most relevant of the issues discussed here, the semantics of communication directly affect the way clustering is done. Any model of clustering needs to take into account the types and semantics of the communicational mechanisms provided in the programming paradigm.
- *Inheritance Vs Prototype:* The mechanism for sharing code/responsibility affects the determination of relationships between objects. Inheritance argues for an additive model of metrics. As subclasses may add features which extends a base class, so too there is a possible extension to the interaction with other objects. Prototypical object models push for more dynamic clustering techniques as object types can not generally be infered statically.
- *Synchronization mechanisms:* Although synchronization does not directly affect the clustering model, it does impact on static analysis mechanisms and the ease with which the model can be applied to automate the process.

Granularity: Since the need for clusters arises out of a need to bridge the granularity gap existing between the programming and run-time systems, cluster granularity must be a free parameter. The questions which need to be answered are: Is there a minimum size for a cluster? Is there a maximum size? Are these values dependent on the set of target architectures? Large cluster sizes result in higher cost of swapping/mapping clusters in and out of memory as well as a higher cost associated with mobility (migration). Small cluster sizes may result in more "cluster faults" and more "cross-context" communication. Cluster size also impacts run-time management issues such as name management.

Number of Clusters: The number of available resources acts as at least a mapping constraint if not directly as a constraint on clustering. Ideally we would like to place each cluster on its own processor.

Metrics: Any form of clustering must be based on measurable or estimable quantities indicating those relations between objects that lead them to be colocated, as well as goodness criteria for evaluating the resulting clusters. In practice, no set of metrics can lead to *optimal* clustering, in the sense of maximizing total expected application performance. Even if the lifetimes, CPU and storage requirements, interaction patterns of all objects, resource capacities, and communication latencies of all possible clusters were fixed and known, optimal assignment remains an *NP-complete* bin-packing problem.

Automaticity: User level clustering cannot take advantage of the type of dynamic information available to the run-time system, e.g., specific resource availability. Relying entirely on automated clustering prevents taking advantage of designer knowledge which may lead to superior manual solutions. Another related issue is the ease with which a given methodology can be automated. Can it be implemented purely with the aid of static analysis? How dependent is the model on information which is only available at run-time? How do approximations of run-time activity affect the final clustering?

Architectural Dependence: Targeting clusters for a specific architecture is commonly called grain size adaptation. An interesting issue here is "should a clustering strategy attempt to isolate adaptation from partitioning?" Clustering/partitioning should be based solely on the program's characteristics and the subsequent process of mapping to a particular target architecture should perform grain size adaptation. This can/should be structured as a hierarchical process in which the adaptation is often done dynamically.

Another architectural issue is one of modeling communication facilities between processors. There are a number of options:

- The cost of inter-module communication is independent of the processors to which the modules are assigned.
- The communicational cost is dependent only on whether the modules are on the same processor or not.
- The cost is dependent on the configuration and communication costs directly depend on which processors the modules are assigned to. This covers the case of a specific network and can be further refined to take network load into account.

4.3 Resource-Based Clustering Model

4.3.1 Assumptions

Although the ultimate goal of this effort is to map applications to heterogeneous distributed architectures, the models focused upon here consider a simplified architectural model. The simplifying assumptions are:

- The network is an "equal cost" homogeneous network. By this we mean that any two nodes are assumed to be directly connected and there exists a fixed latency for message delivery from one node to another.

- We also do not take architectural and software heterogeneity into account for now.
- Asynchronous invocations release the sending object as soon as the message is queued at the receiver.
- There exists a fixed "location latency" for locating any object on the system.

These assumptions do not in any way affect the model of clustering which we are about to present and it can be extended to take all of these factors in account without significant effort.

4.3.2 Basis Model

We propose a model of cluster formation that may be applied regardless of whether or not the process is automated. However, the model lends itself well to automation. Cluster formation within the DIAMONDS environment is based on static analysis of active object code to implicitly detect the underlying concurrency among classes and objects. However our approach is to restrict concurrency in a structured manner. More specifically, the model and subsequent clusterization tool detects objects which are likely to be tightly coupled in synchronous interactions.

Classical parallel processing approaches to exploiting concurrency involve the partitioning of data/computation so as to allow more than one processor to share the load of the problem. Increasing the number of processors beyond a certain point produces no additional speedup and at some point the speedup curve will decrease. There are several reasons for behavior. One is the fraction of sequential computation, the effects of which serve to place an upper bound on speedup according to Amdhal's law. Second, at some point, for realistic communication networks, the overhead for interprocess communication and synchronization overwhelm the benefit of additional computing resources. In loosely coupled systems the effect of communicational costs plays a much more important rôle than in shared memory systems.

In a fine grained active object model, every object is potentially active and can be placed in a process by itself. In this model there is concurrency available which cannot be utilized by the application. Thus, the physical design task necessitates mapping multiple objects to a process/processor so as to reduce communication overhead and utilize a number of processors which is closer to that which can be exploited by the application. This is a process of composition where active objects are passivated by bringing them together wherever sensible.

Note that with respect to the discussion presented here, communication between modules depends only on whether the modules are on the same processor or not and inter-processor communication metrics (latency and bandwidth) are the same for any two processors. Network loading is not considered. These factors will affect the performance, expanding the model (to include more realistic network topologies) should not be a significant change and will be considered in the future.

The Approach

The DIAMONDS infrastructure is multi-layered with support for object clustering coming from many sources. At the uppermost layer, the developer is presented with a programming model

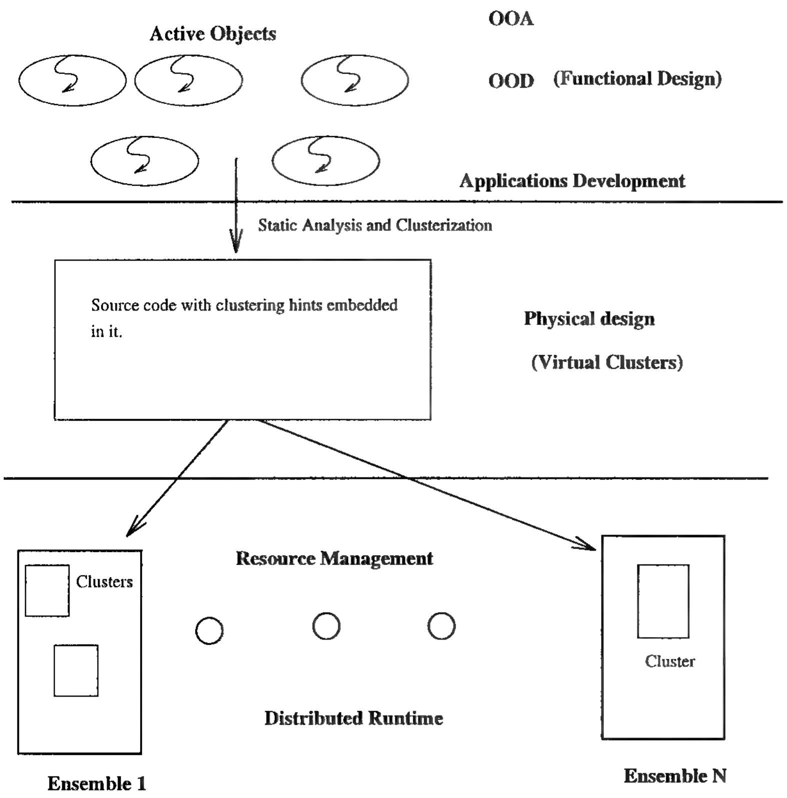

Figure 5: The Tiered Approach of DIAMONDS.

based on fine-grained active objects and the facilities for the explicit expression of clustering hints. The prgrams are then statically analyzed to identify further restrictions to internal concurrency. This analysis leads to a modified program with additional clustering hints embedded within it. Dynamic composition, utilizing the hints, maps the application onto the target architecture. Resource management makes modification to the mapping as needed. This tiered approach is shown in Figure 5. The emphasis of the following few sections is on the initial parititioning also termed clustering and a model for accomplishing the same.

The Object Graph & Affinities

The basis for clustering is a weighted, undirected object graph in which the nodes represent objects and the arcs represent object interactions. The nodes are weighted to reflect the complexity of the object while the arcs weight indicates the frequency of interaction. Since synchronous interactions and asynchronous interactions have distictively different implications on the concurrency between objects, the arcs are labeled with two weights one for each type of potential interaction. These arc weights are called *affinity tuples.*

There are two dimensions to any analysis resulting in affinity tuples:

1. *Static* versus *Dynamic.* Static measures are those reflecting relationships between objects that hold, or potentially hold, sometime across the lifetimes of the associated objects. Dynamic measures summarize affinities based on connection and interaction patterns that hold at particular points in the lifetimes of objects.

2. *Class* versus *Instance.* Class-based analyses relate all instances of one class to all instances of another, without regard for the fact that the relations may, for example, only hold between certain special pairs of instances. Instance-level analyses relate particular objects.

While dynamically-based instance-level measures provide the best guidance, they are often difficult or impossible to measure. Because new objects may be constructed at any point, for any reason, during the lifetime of a system, it is normally impossible to track all of them. Thus, class-based criteria play a large role in analysis. Many class-based analyses are in turn static.

These measures lead to definition of aggregate affinity tuple A_{ij} between any two particular objects O_i and O_j, where O_i is an instance of class C_x and O_j is an instance of class C_y. These affinity tuples, $< A^S_{ij}, A^A_{ij} >$, are comprised of a synchronous interaction element and an asynchronous interaction element, respectively. Further, each tuple element is additively constructed from its per-instance affinity tuple element, IA_{ij}, and its class-based affinity tuple element, CA_{xy}, resulting in:

$$< A^S_{ij}, A^A_{ij} > \;=\; < IA^S_{ij}, IA^A_{ij} > + < CA^S_{xy}, CA^A_{xy} > \qquad (1)$$

In the resulting framework higher synchronous affinities represent stronger evidence for colocation whereas higher asynchronous affinities suggest a resistance to colocation. Affinities are additive through subclassing – a subclass adds affinities to those established in its superclass(es). $CA_{Any,Any}$, where *Any* is the root of an inheritance hierarchy, may be set as <zero,zero>.

Each class-based element of an affinity tuple is composed from:

1. a *probabilistic* component, S_{xy}, denoting the probability of interaction between the respective classes.

2. and a corresponding *strength* component D_{xy} indicating the strength of interaction between the classes, given the fact that the classes do interact.

The total inter-class affinity can then be written as:

$$CA_{xy} \quad = \quad < S^S_{xy} \times D^S_{xy}, S^A_{xy} \times D^A_{xy} > \tag{2}$$

The following sections discuss a model for determining each of these components.

Interaction Probabilities The approach taken here is a refinement of one of the heuristics listed in [CR92], rephrased in a way that gets at the heart of the primary results of clustering, *single-threadedness*, and serves as a static approximation of the dynamic analyses described below. When one object synchronously invokes another, it requires no compute resources while waiting for the result. Minimizing wait time simply entails minimization of communication latency, and thus maximizing the likelihood that the two objects reside in the same address space and cluster. Asynchronous invocations represent an increase in concurrency, however, there is a synchronization penalty associated with the sender blocking until acknowledgement of delivery.

$$\begin{aligned} P_s(C_x \rightsquigarrow C_y) &= P(C_x \; is \; referenced \Rightarrow C_y \; is \; referenced \; synchronously) \\ &= \frac{\sum_{method \; m \in C_x} \# \; of \; sync \; invocations \; on \; C_y}{\sum_{method \; m \in C_x} total \; \# \; invocations} \qquad (3) \\ P_a(C_x \rightsquigarrow C_y) &= P(C_x \; is \; referenced \Rightarrow C_y \; is \; referenced \; asynchronously) \\ &= \frac{\sum_{method \; m \in C_x} \# \; of \; async \; invocations \; on \; C_y}{\sum_{method \; m \in C_x} total \; \# \; invocations} \qquad (4) \end{aligned}$$

This can be determined by the considering the code of each method of class X and looking for messages to objects of class Y. A refinement to this approximation could be made by considering this on a per method basis and considering the actual control flow within methods of C_x. Then these probabilities (of communicating with C_y from some method m) could be weighted by the relative probability of C_x being "active" executing the particular method code (m).

The weighting factors that are then applied in determining the affinity tuples can then be considered to be the result of ORing the two events responsible for the interaction between the classes. The probability of an event which is the OR of two events has been taken to be the total interactional probability of the pair of classes. This is because we consider interaction measures to be bidirectional and our "object graph" to be undirected.

$$S^S_{xy} \quad = \quad P_s(C_x \rightsquigarrow C_y) + P_s(C_y \rightsquigarrow C_x) - P_s(C_x \rightsquigarrow C_y) \times P_s(C_y \rightsquigarrow C_x)$$

$$S^A_{xy} = P_a(C_x \rightsquigarrow C_y) + P_a(C_y \rightsquigarrow C_x) - P_a(C_x \rightsquigarrow C_y) \times P_a(C_y \rightsquigarrow C_x) \qquad (5)$$

A quantitative analyses of each invocation results in complementary strength factors, D^*_{xy}. The strength factor represents the cost incurred if the dynamic configuration is incompatible with the type of communication, i.e., for a synchronous interaction the objects are contralocated and for an asynchronous interaction the object are colocated.

Given this notion, we have that D^S_{xy} is the cummulative blocking (synchronization) time associated with a single synchronous interaction, while D^A_{xy} represents the lost concurrency. Let:

M_x = the set of methods of X.
$I(\alpha)$ = invocation time for method α.
$N_{xy}(\alpha)$ = frequency at which methods of X invoke method α of Y.
$C(\alpha)$ = *Complexity coefficient* of method α. An estimate of a method's execution time.

Consider an object o_i invoking a method on object o_j. Then the per invocation penalties may be considered as follows:

1. *Synchronous:* In synchronous invocations, the sender blocks, waiting for a reply. The time for which it is blocked depends on both the time of dispatch as well the complexity of the invoked method. Hence cost of invoking α synchronously is given by:

$$D_s(\alpha) = I(\alpha) + C(\alpha) \qquad (6)$$

2. *Asynchronous:* Although asynchronous invocations by definition are one-way sends, we need to consider the time required for dispatch before the sending object is freed. (This time is negligible if the two objects are colocated). The complexity of the invoked method represents the lost concurrency which must by offset by the invocation overhead. Thus, short aynchronous methods do not represent a significant repulsive force between two objects.

$$D_a(\alpha) = C(\alpha) - I(\alpha) \qquad (7)$$

The invocation cost I depends on primarily on communication delay and message marshalling, and demarshalling overhead. Note, a method which requires a large amount of data transfer will incur a larger invocation cost than one with little or no data transfer.

The strength factors can now be computed [1]. We may now define $F_s(X \rightsquigarrow Y)$ and $F_a(X \rightsquigarrow Y)$ as the directed strength factors where an instance of X potentially invoking methods on an instance of Y.

$$\begin{aligned}
D^S_{xy} &= F_s(X \rightsquigarrow Y) + F_s(Y \rightsquigarrow X) \\
&= \sum_{\mu \in M_y} N_{xy}(\mu) \times D_s(\mu) \\
&+ \sum_{\alpha \in M_x} N_{yx}(\alpha) \times D_s(\alpha) \qquad (8) \\
D^A_{xy} &= F_a(X \rightsquigarrow Y) + F_a(Y \rightsquigarrow X) \\
&= \sum_{\mu \in M_y} N_{xy}(\mu) \times D_a(\mu) \\
&+ \sum_{\alpha \in M_x} N_{yx}(\alpha) \times D_a(\alpha) \qquad (9)
\end{aligned}$$

Complexities What remains to be determined is the complexity factor $C(\alpha)$ of a method α. This essentially represents the time complexity in terms of how long the method takes to execute. It could be cast in terms of the number of instructions in the intermediate code appropriately weighted by the presence of loops and calls to other methods.

Usage - The Third Dimension The model does not consider the way in which a class is used. As was indicated previously, the probabilistic measures could be analyzed on a per method basis, i.e., $P_*(C_x, m_i \rightsquigarrow C_y)$, the probability of a interaction with class C_y from method m_i of class C_x. If this approach is taken then the frequency of occurence (probability of invocation) of m_i being active, depends on the set of objects interacting with an object of class C_x. It is quite common for an object to have only one "user". In those cases, a more accurate class-based interaction affinity tuple (directed) could be computed by restricting the set of methods in C_x which are considered to interact with objects of class C_y due to the influence of the "user", C_z, of C_x. Consider the example denoted by figure 6. In this example the situation depicted is one where the affinity between classes `Server1` and `Server2` is not independent of the rest of the system but instead depends on the user class we are considering. With respect to class `User1`, `Server2` interacts purely synchronously with `Server1` while with respect to class `User2`, `Server2` interacts purely asynchronously with `Server1`. Hence the synchronous affinity element

[1] We assume that synchronicity is a property of the method and not of the invocation.

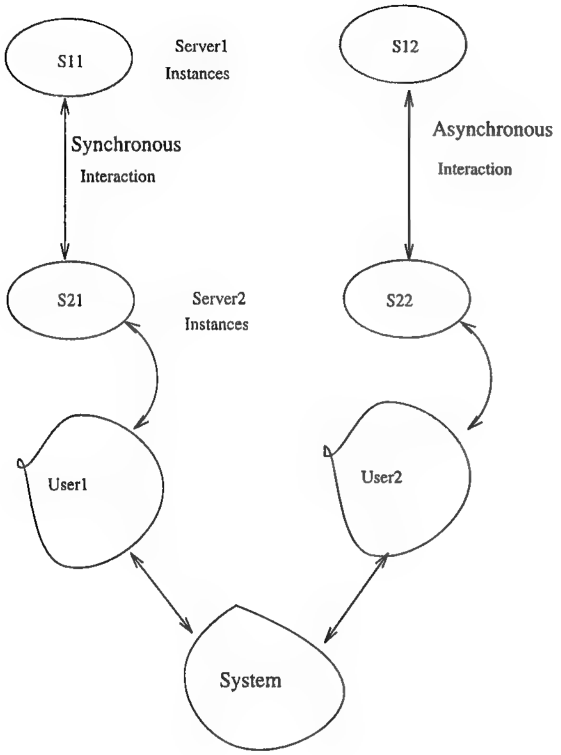

Figure 6: Differing Usage Patterns.

in the first case would be high, whereas in the latter case the asynchronous affinity element would be high.

This type of dependency can be modeled as the addition of a third dimension to the affinity matrix. This dimension would simply indicate the class with respect to which we are considering the affinity factor. It would account for the usage pattern of a particular class by another. Our affinity calculations can now be modified to include the effect of usage patterns by simply taking into account the number of times a particular method being analyzed is called by the class with respect to which we are determining the affinity factor. For purposes of illustration we will work out the equations for access probabilities. Let $U_z(\alpha)$ be the usage factor of method $\alpha \in C_x$ by methods of C_z. Then we have:

$$P_s(C_x \rightsquigarrow C_y \; wrt \; C_z) \;=\; P(C_z \, references \, C_x \Rightarrow C_y \; is \; referenced \; synchronously)$$

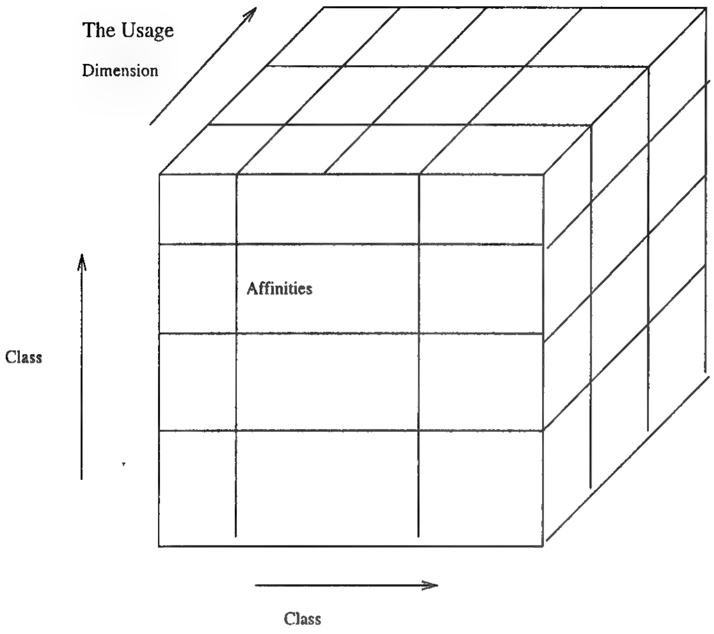

Figure 7: Modeling Usage Via the Third Dimension

$$
\begin{aligned}
&= \frac{\sum_{m \in C_x} U_z(m) \times \#\ of\ sync\ invocations\ on\ C_y}{\sum_{m \in C_x} U_z(m) \times total\ \#\ invocations} && (10)\\
P_a(C_x \rightsquigarrow C_y\ wrt\ C_z) &= P(C_z\ references\ C_x \Rightarrow C_y\ is\ referenced\ asynchronously)\\
&= \frac{\sum_{m \in C_x} U_z(m) \times \#\ of\ async\ invocations\ on\ C_y}{\sum_{m \in C_x} U_z(m) \times total\ \#\ invocations} && (11)
\end{aligned}
$$

Of course, we could just as easily encounter situations which defy this solution, i.e., one in which a user class causes two other classes to interact both synchronously and asynchronously via different methods or behaviors. Class based affinities cannot resolve this problem. What is required is the tracking of object-object interactions and the methods actually invoked at run-time by profiling the application and then suitably altering inter-object affinities before the application of clustering algorithms.

Using Affinities for Clustering

Once we have determined the affinity matrix we need to apply an algorithm for partitioning the graph of objects into a set of clusters. These algorithms are commonly graph partitioning algorithms or statistical clustering algorithms. Using this model still requires the affinity tuples and weights for all objects in the system. This in turn implies knowledge about all objects that are going to exist in the system at run-time. When applying this model to automate the clustering process **via static analysis** it will not likely be possible to statically determine what objects are going to exist at run-time. Of course, a "complete" object-graph can be constructed from profile data and clustering can be performed from this information.

One approach for complete static clustering is to divide the clustering process into two steps. First, consider all objects which are known statically to exist. Proper clustering of these central, "global" objects are vital to the success of any static strategy. Next, consider all occurences of object construction and build an pseudo-object graph. Clustering of this graph, in which most nodes refer to a set of potential objects, will define sets of clusters and relate them to the established "global clusters". A run-time infrastructure must be established which can support these "cluster-sets". Such support, at a minimum, must aid in the cluster creation and object binding to the new clusters.

Another approach, which has been taken by this project, is to consider the point of control which can be provided to a run-time infrastructure by a static clustering tool. For dynamically constructed objects (say via a `new` type statement), the run-time must determine placement at the time of construction. Statically, we can determine a small set of object with which the newly created object interacts (determined by the scope at which the object is created). These objects are represented by references which again stand for potential sets of objects. Hence, we could cluster/partition this set and thus determine the relative placement of the "new" object with respect to the other references in this set. This relative placement holds for the set of potential objects that the "new" represents. This is explained in greater detail in the following sections on static analysis.

4.3.3 Static Clustering

The fundamental issue in static analysis of object oriented languages is that while affinities (and/or other metrics) exist between *actual* objects, a program mainly specifies *potential* objects. Classes describe families of objects that may be constructed in the course of execution. Sometimes only a few objects are statically indicated to always exist at run-time. This problem is compounded by the fact that in some program contexts, objects are known only by their superclass types, not their maximal (most exact) class membership. While these problems do limit the precision of static analysis, they by no means eliminate its practical utility. Affinity measures represent the only plausible *a priori* means of predictive guidance for run-time placement. Given the orders-of-magnitude throughput differences that can result from bad placements and dynamic reclustering, successful static analyses that reduces unnecessary cross-cluster messages by only a few percent can have a dramatic effect. In some cases they might do much better, all but eliminating the need for run-time backup. In other cases, affinities may be improved via run-time monitoring of interactions and placements, coupled with dynamic reclusterization.

The Methodology

This effort performs *pseudo-static* clustering for ODL source. A brief description of each of the modules of this process and their functions is furnished below:

Parser: The parser builds up an internal representation of the input ODL program and presents the same to the rest of the system as an Abstract Syntax Tree (AST) (See section 3.1.1 and 3.1.2).

Static Analyzer: This module analyzes the AST representation by traversal and builds up an *affinity matrix.* A key assumption that we make at this stage is that inter-class affinities are representative of the behavior of their corresponding instances since it is impossible to statically predict the existence and behavior of actual instances. Thus the affinity matrix is one in which entries represent inter-class affinities. In building an affinity matrix we also accomplish individual class analysis.

Cluster Analyzer: The main function of this module is to analyze the affinity matrix to determine the number of clusters intrinsic to the design. This can then be fed as a parameter to the clustering algorithm outlined in the previous chapter. Although this is part of the "idealized" clustering process, we have not included this as a part of the current prototype.

Clusterizer: Forms a set of clusters out of the affinity matrix by applying various clustering algorithms. In effect it determines the placement of the instance associated with each "new" (dynamic creation) relative to other instances.

Assimilator: Encapsulates the responsibility of feeding the clustering information back into the AST in a manner which can eventually be used by the run-time system for the placement of objects on creation.

For the clustering tool prototyped for this effort a simpler class graph and affinity measures were developed. The nodes are unweighted (although complexity plays a role in the affinity measures), and only synchronous interaction metrics are considered. (Actually, asynchronous interactions indicate a low affinity to colocation which implies contralocation in the absence of significant synchronous interactions). Once this affinity matrix has been obtained the code is examined to determine the places where new objects are constructed and appropriate placement hints for these objects are added. The algorithm that we employ is shown below and assists in the placement of a newly created object with respect to the locations of other objects in the system. The set of objects which is accessible to, and which can access the new object forms the candidate set in our case. Note that in the proposed algorithm we place one object

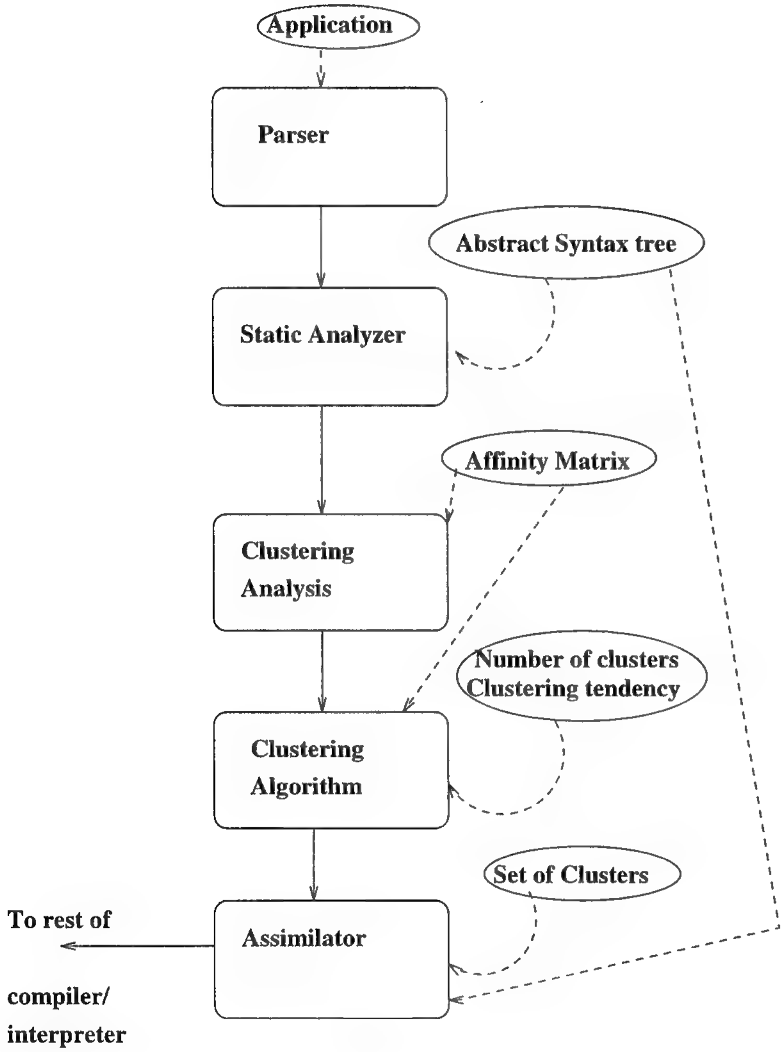

Figure 8: The Typical Clustering Process

Notations Used

Let

* o_n *be the object being newly constructed.*

* O *be set of objects that* o_n **potentially** *interacts with.*

* Ψ_p *be the threshold value of probability of synchronous communication for clustering.*

* Ψ_a *be the threshold value for ratios of affinities for clustering.*

* C_i *be the class of* o_i $\forall o_i$

* o_{ct} *be the object which constructs* o_n

* M *be the method in which* o_n *is constructed.*

* o_s *be the* **sender** *object which calls* M.

* $CTV(FFV)$ *be integers denoting the number of votes in favor of a placement of* o_n *close to(far from) the object under consideration.*

* $CTList(FFList)$ *be the close to(far from) lists for* o_n.

* $\forall(C_i, C_j)$, *let*

> * $A_{i<->j,k}$ *be the affinity between* C_i *and* C_j *with respect to class* C_k.
>
> * $P_{i<->j,k}(S)$ = *probability of synchronous communication between* C_i *and* C_j *with respect to* C_k.
>
> * $P_{i<->j,k}(A)$ = *probability of asynchronous communication between* C_i *and* C_j *with respect to* C_k.

Figure 9: Notations Employed in the Algorithm.

at a time in the sense that each new is annotated separately. This can also be interpreted as an approach wherein we partition the local object graph corresponding to the method thereby packing objects encountered into clusters.

The basic premise on which this algorithm is built is that there exists a set of objects which can potentially interact with the newly created object whose placement is to be determined. This set is a union of several categories described below. There are several parts to this algorithm. Each part consists of determining placement relative to a particular category of object such as constructing object, sending object and so on. We will take the approach of describing the overall algorithm and then each of it's constituent parts will be expanded where necessary. Figure 9 describes the set of notations used in the algorithm.

Main The main procedure is shown in figure 10. It acts primarily as a driver splitting up the set of all relevant objects accessible by/from the new object being created into the following categories of objects:

- A **creator** object o_{ct} which is the object on whom the method M is called. The new object is created in M. A procedure called *Creator* is invoked which determines placement of o_n relative to o_{ct}.
- A **sender** object o_s which is the object responsible for calling M on o_{ct}. The driver invokes a procedure called `Sender` to determine placement relative to o_s.
- The set of all objects excluding o_{ct} and o_s which can potentially interact with o_n. This set is a union of:
  - $\{o \mid o\ is\ a\ local\ of\ M\}$,
  - $\{o \mid o\ is\ a\ attribute\ of\ C_{ct}\}$ and
  - $\{o \mid o\ is\ a\ global\}$

  As far as locals are concerned we need to take into account the scope at which a `new` is seen. For the second category we are considering a flattened version of C_{ct} since base class attributes are visible to the derived classes unless explicitly forbidden using protection domains. This aspect is language dependent and for the algorithm we will use a generic definition of "all available attributes".

Creator The procedure *Creator* shown in figure 11 is primarily concerned determining the relationship of o_{ct} with o_n. This requires considering the nature of the link to which o_n is bound. Let l_n be the link to which o_n is bound. Two cases arise here:

1. l_n is local to the method M. In this case we need examine only the interactions initiated by o_{ct} in M on o_n after it's construction. There is however, an additional point to be considered here, i.e., whether o_{ct}'s identity is exported to o_n in any of these interactions. If so, then we should consider the affinity between C_{ct} and C_n also since we cannot determine how o_n might use o_{ct}'s identity. If not, the first part of the analysis is enough to determine placement. Also note that we give additional weightage to the creator's interaction with o_n if o_n is local to the class creating it.
2. l_n is accessible by all methods of C_{ct}. This can happen in one of 3 ways:
   - It is an attribute of C_{ct}.
   - It is declared in M's scope and exported.

```
Procedure placeObject
        // Determine relationship to $o_{ct}$
        call procedure Creator.
        // Determine relationship to $o_s$
        if $o_n$ is ExportedToSender then
            call procedure Sender.
        endif
        // Determine relationship to all $o_i \in O$
        // such that $o_i \neq o_{ct}$ and $o_i \neq o_s$
        call procedure Others.
        // Postprocess to determine placement.
        call procedure Postprocess.
```

Figure 10: Main Procedure of Placement.

- It is a globally accessible link.

In this case all methods $\in C_{ct}$ have to be analyzed for invocations on o_n(using l_n and it's aliases). Again if the constructing objects identity is exported in an invocation on o_n we also consider the affinity between the two classes involved in determining placement.

Clearly in all cases high relative affinities and predominantly synchronous interactions cause placement close to the constructing object.

Sender This procedure shown in figure 12 determines the relationship of o_n to o_s - the object calling method M on o_{ct}. It is invoked only if the identity of the newly created object is exported explicitly by one of the means already defined to the sending object. Since we cannot statically determine the identity of the sender object, we provide a location hint relative to the sender only if the vote is overwhelmingly in favor of colocating or contralocating. If the relationship is not conclusive with regard to placement, we defer consideration of placement relative to sender till run-time. This part of the procedure is indicated in procedure **Postprocess**.

Others As has been indicated the universe of objects which o_n can interact with is a union of the creator, the sender, the locals declared in the method M, the attributes of the class C_{ct} and all globally declared objects. This procedure deals with defining relationships of all objects apart from the sender and creator with o_n.

```
Procedure Creator
        $CTV := FFV := 0$
        Let $S_m$ be the set of methods we need to analyze.
        Initially $S_m = \{\}$
        Case: $o_n$ accessible within $M$ only.
            $S_m = S_m + \{ M \}$
        Case: $o_n$ can be accessed by all methods of $C_{ct}$.
            $S_m = S_m + \{m \; / m \in C_{ct}\}$
        End Cases

        $\forall$ invocations on $o_n$ in each $m \in S_m$
        If $\frac{sync}{sync+async} > 0.7$ Then
            If $o_n$ is local to $Creator$ Then
              Increment $CTV[Creator]$ by 2
            Else
              Increment $CTV[Creator]$ by 1
            EndIf
        ElseIf $\frac{async}{sync+async} > 0.7$ Then
            If $o_n$ is local to $Creator$ Then
              Increment $FFV[Creator]$ by 2
            Else
              Increment $FFV[Creator]$ by 1
            EndIf
        EndIf

        If $o_{ct}$ exported to $o_n$ Then
            $S_m = \{m \; / \; m \in C_n \}$
            For each $C_i$ invoking one or more $m \in S_m$
              If $P_{n<->ct,i}(S) \geq \Psi_p$ and $\frac{A_{n<->ct,i}}{MAXAFF} \geq \Psi_a$ Then
                Increment $CTV[Creator]$
              ElseIf $P_{n<->ct,i}(A) \geq (1 - \Psi_p)$ Then
                Increment $FFV[Creator]$
              EndIf
            EndFor
        EndIf
```

Figure 11: Procedure Creator of Placement.

```
Procedure Sender
        Let $S_c$ be the set of classes of the possible senders.
        $S_c = \{C/P(C \text{ invokes } M) > 0\}$
        For each $C_i \in S_c$ do
            If $P_{n<->ct,i}(S) \geq \Psi_p$ and $\frac{A_{n<->ct,i}}{MAXAFF} \geq \Psi_a$ Then
                Increment $CTV[Sender]$
            else if $P_{n<->ct,i}(A) \geq (1 - \Psi_p)$Then
                Increment $FFV[Sender]$
            EndIf
        EndFor
```

Figure 12: Procedure Sender of Placement.

Since we have no additional information regarding these objects' interactions with o_n, we consider only class-based affinities here.

For each object belonging to the set of "other" objects we essentially consider it's affinity (class-based) relative to the maximum possible affinity of C_n with any other class such that an instance of that class belongs to O.

Post Processing Once the relationship between o_n and other objects has been derived, placement hints need to be determined. These are determined primarily using the close to and far from votes described earlier.

Ratios of the votes with respect to the total votes polled help decide on a particular placement. Is no decision can be arrived at eventually, i.e., both the lists are empty, a hint to create a new cluster and place o_n in it is inserted. The sender's case is special as we need a near unanimous vote on a particular placement as the sender's identity is now known. Only if we can be sure that regardless of the type [2] of the sender a particular placement will not result in degradation of runtime performance, we can suggest a placement. Else we defer the decision to place with respect to this object to run-time for lack of information. Other points to note here are:

- We eventually rank order the lists in order that the run-time can make quick decisions in the event of a conflict.

- We also retain the degree of "closeness" - which is the percentage of CTV and "repulsion"

[2] By type, we mean it's abstract type as represented by it's abstract class.

```
Procedure Others
        let $O := O - \{o_{ct}, o_s\}$
        For each $o_i \in O$ do
            Let $S_c = \{C \;/\; C$ can cause an interaction between $o_i$ and $o_n\}$
            For each $C_j \in S_c$ do
                If $P_{n<->i,j}(S) \geq \Psi_p$ and $\frac{A_{n<->i,j}}{MAXAFF} \geq \Psi_a$ Then
                    Increment $CCV[o_i]$;
                Else If $P_{n<->i,j}(A) \geq (1 - \Psi_p)$ Then
                    Increment $FFV[o_i]$;
                EndIf
            End For
        End For
```

Figure 13: Procedure Others of Placement.

- which is the percentage of FFV for each candidate object. This information can also be helpful in the event of a conflict.

Annotations The function of the assimilator module in figure 8 is to insert hints (run-time directives) regarding object construction into the intermediate representation. Every time an object is constructed it is assumed that the run-time constructs the cluster that the object is to be placed into and creates a new cluster if required. Hints are annotations to the `new` statement that take the form:
new `className (arg1.....argN)` *hint*
The hint can be in one of three forms:

- `close-to` $< ObjID, ..., ObjID >$ This is interpreted as follows: *Place the object under consideration as close as possible to any one of the list of* $< ObjID >$ *in the order: same cluster, same ensemble, same node.* The list contains those objects with the highest affinities, as determined by a threshold. When possible, the list is reduced to a single element.

- `far-from` $< ObjID, ..., ObjID >$ This means: *Place the object under consideration as far away as possible from the list of* $< ObjID >$ *in the order: different node, different ensemble, different cluster.* For either of these two types of hints the first choice in the order shown is chosen and if the placement is not possible because of size constraints then we move down the list.

```
Procedure Postprocess
        let $O := O - \{o_s\}$
        For each $o_i \in O$ do
            If $\frac{CTV[o_i]}{CTV[o_i]+FFV[o_i]} > 0.6$ Then
              append $o_i$ to $CTList$
            Else If $\frac{FFV[o_i]}{CTV[o_i]+FFV[o_i]} > 0.6$ Then
              append $o_i$ to $FFList$
            EndIf
        End For
        // Deal with the sender, as it is a special case.
            If $\frac{CTV[o_s]}{CTV[o_s]+FFV[o_s]} > 0.9$ Then
              append $o_s$ to $CTList$
            Else If $\frac{FFV[o_i]}{CTV[o_i]+FFV[o_i]} > 0.9$ Then
              append $o_s$ to $FFList$
            Else
              defer placement wrt sender to run-time
            EndIf
            Rank $CTList$ in decreasing order of closeness.
            Rank $FFList$ in increasing order of closeness.
            If empty($CTList$) AND empty($FFList$) Then
              place $o_n$ in a new cluster
            End If
```

Figure 14: Procedure Postprocess of Placement.

- `Sender` Interpreted to mean that the runtime needs to determine the sender and decide the relation of the new object to the sender. A clear placement hint could not be obtained statically.

- `new asseed` Interpreted to mean *always create a new cluster and place this object in it.*

Note that *ObjID* could be the link corresponding to either the sending object, the constructing object or any other object which can be accessed from the scope in which the `new` statement exists. We could also have any combination of the first three types as long as there are no contradictions in that:

- An *ObjID* cannot appear in both the `close-to` and `far-from` lists.

- A Sender hint would imply that we cannot statically determine placement with respect to the sending object and hence the sender object cannot be in either of the two lists.

Run-Time Considerations When a new object is created, the run-time evaluates the hints. This could result in a situation requiring further analysis, when the hints were precise enough to make the placement. Situations where this could happen are:

- When we cannot resolve a relationship with the sender statically and the newly created object's identity *is* exported to the sender. In this case the run-time needs to evaluate the relative affinities of the sender and o_n dynamically and make a decision.

- When a contradiction appears to exist in the hints. This can happen when:

  - $x \in CTList$, $y \in FFList$ and x and y happen to be in the same cluster.
  - $x \in CTList$, $y \in CTList$ and x and y happen to be in different clusters.

  In these situations the run-time needs to break ties depending on the relative degree of closeness. This can be done with the help of the differential in the votes which are retained.

- When the creation of a new cluster is required. This can happen in situations such as:

  - If the close-to list is empty, there could arise a situation where processing the far-from list implies that no cluster is available to place o_n in. In these cases a new cluster will have to be created.
  - There is a dynamic exception raised due to a cluster exceeding its maximum size. This is a parameter built into the system and is dependent on page size and other architectural features.

- Lastly it must be realized that evaluation of the hints is a costly process in itself requiring processing resources. To reduce this cost there will exist a default case without any hint. This can be treated either as a new cluster creation or a case of placement close-to the creating object. Unless a strong indication exists for a particular hint, the default, low cost new, will be used.

4.4 An Example

We will illustrate the methodology with the help of a simple example.

```
class Server1
  op m1 {
      % asynchronous complex method.
  };

  op m2 : () {
        % synchronous complex method.
  };
end;

class Server2
  op syncWithServer1(s:Server1){
        % Frequent synchronous interactions with Server1.
        % Invokes m2 repeatedly.
  };

  op asyncWithServer1(s:Server1){
        % Frequent Asynchronous interactions with Server1.
        % invokes m1 repeatedly.
  };
end;

class User1
        s1: Server1 <>;
        s2: Server2 <>;

        op work:() {
                % Creates new Server1 and Server2 objects.
                % Invokes syncWithServer1 on s2 thereby causing
                % synchronous interaction between s1 and s2.
        };
end;

class User2
        s1: Server1 <>;
        s2: Server2 <>;

        op work:() {
                % Creates new Server1 and Server2 objects.
                % Invokes asyncWithServer1 on s2 thereby causing
                % asynchronous interaction between s1 and s2.
        };
end;

op main {
        % The driver op which runs the program
        local u1:User1 = new User1;
        local u2:User2 = new User2;

        u1.work();
        u2.work();
}
```

The example shown uses the minimal set of features of ODL. It features two server classes whose objects interact with each other. For the sake of simplicity we have retained pure communication modes here. There are two user classes which cause the two specific types of interactions between the servers. A driver method main controls the computation. The output of the analyzer is a set of statements showing the creation of new objects and their placement hints which appears as follows:

```
Class User1 Attributes
        s2      local
        s1      Exported
        op work:
                s1: new Server1 new cluster
                s2: new Server2 farFrom creator, closeTo s1

Class User2 Attributes
        s2      local
        s1      Exported
        op work:
                s1: new Server1 new cluster
                s2: new Server2 farFrom creator, farFrom s1

Class System Attributes
        op main:
                u1      local
                u2      local
```

```
u1: new User1 closeTo creator
u2: new User2 closeTo creator
```

4.5 Validation

Having presented the theory of clustering and a strategy to apply this theory in prototyping a static analysis tool, we now propose a methodology with which we can validate the effectiveness of the same. There are a number of problems associated with trying to validate the proposed methods. A significant one is the lack of applications which have been written based on an active object programming model. Another problem is the lack of appropriate metrics by which one can gauge how well the model works. We first present a set of metrics and baseline cases against which our model will be validated and then the actual experimental results.

4.5.1 Validation Metrics

There exist two forms of the verification both of which can be used to verify the overall validity of the process, one *external* and one *internal*. Each of these forms have different cost functions which need to be evaluated. The former seeks to determine a range for number of clusters which yield acceptable throughput. The latter, internal validation, seeks to verify the goodness of partitioning given the number of clusters.

External Validation

External validation seeks to verify the model without relying on the products or the process that was used to create the partition in the first place. In other words, this method of verification uses absolute measures of system performance such as execution time as a validation metric. External criteria measure performance by matching a clustering structure to a baseline clustering which exhibits a known type of behaviour. In our case we have two baseline clusterings, one in which we have a single cluster and the other a situation in which every object lives in it's own cluster.

The set of metrics which act as external validation measures are:

Response Time: $R(C)$ is the total number of global cycles for which the application executes and is the maximum of the cluster lifetimes over all clusters. Clearly, a lower response time indicates better performance. $R(C_s)$ denotes the response time for the case with C clusters and structured placement while $R(C_r)$ denotes response time for random placement.

Idle Time: I is the fraction of each global cycle that was *not* used in execution averaged over the lifetime of the application and all the clusters. Lower idle times indicate better performance in general although the number of clusters also affects this figure considerably. The reference situation is one in which the idle time is 0 and all objects exist in a single cluster.

Queue Length: Q is the number of contexts waiting to be executed during any global cycle averaged over the lifetime of the application and all clusters. Queue length is related to the idle I. In general one may expect that higher the queue length the lower the idle.

Speedup: $S(C)$ is defined to the ration of the response time with C clusters to the response time with 1 cluster.

$$S(C) = \frac{R(C_s)}{R(1)} \tag{12}$$

A higher value of speedup shows the additional resources employed gainfully and hence is indicative of better performance.

Efficiency: $E(C)$ Efficiency is a measure of the use to which the clusters are put during the lifetime of an application. It can also be seen as a function of the idle time of clusters. The cost function E can therefore be written as:

$$E(C) = \frac{S(C)}{C} \tag{13}$$

Note that efficiency is a function of placement. Two different placements with the same number of clusters may not necessarily yield the same efficiency since random placement may lead to higher cross cluster communication which may not compensate for the added concurrency introduced.

Internal Validation

This is a process of validation wherein the proximity matrix itself is used as a basis for evaluating the effectiveness of a particular partition. No external agents are used. Clearly this is a much simpler procedure and involves significantly less effort than the former one. The cost function to be determined for any given partition P is:

$$M = \frac{\sum_{i=0}^{\#C} \frac{\sum_{o_j \in C_i} inter-cluster\ affinities\ of\ o_j}{\sum_{o_j \in C_i} Intra-cluster\ affinities\ of\ o_j}}{\#C} \tag{14}$$

The smaller this number the better the partitioning. However, because it is based on affinities as part of it's cost function, it is not sensitive to glitches in the model producing these affinities in the first place. It requires the number of clusters to be known *a priori* and will validate only the effectiveness of the partitioning itself. However, it may be assessed with respect to other partitions. The reference partition with which we can evaluate our partition is a random one. It has the same number of clusters, and the same number of objects per cluster, as the original partition which we are trying to evaluate. There exists a number of such random partitions, one of which is in fact the "ideal" one (in that it allows for maximum throughput). Due to the inherent weaknesses of validation using internal criterion we will not consider these methods further and instead expand on the methods using external(absolute) criterion.

4.5.2 Reference Cases

Given a set of objects, determining the "optimal" partition of the set is an NP-complete problem except for a limited situation of very few objects. Even if the number of clusters is fixed there is a combinatorial explosion in the number of possible clusterings for an increased number of objects. For our purposes, we define three such cases:

1. All objects in one cluster. Clearly this is an important reference clustering as it lies at one end of the spectrum of possibilities.

2. One object per cluster. Again, this represents an extreme view, one from which we started. It also reveals the performance of such systems if no form of grain adaptation was utilized.

3. Clustering done by using heuristics and profiled data as per Chatterjee's method [CR92].

4. Random clustering for the number of clusters suggested by the analyzer. This allows us to determine the effect of affinities and object relations as they relate to locality and performance.

4.5.3 Experimental results

In order to verify the effectiveness of the model and analytical tools we have built a simulation of a distributed object execution system. The simulator has modes which allows one either to control placement manually or can perform automatic placement. The latter form of placement can be directed to be random or according to the clustering hints provided by our analyzer.

A commonly used benchmark, the nqueens problem is one of finding as many solutions as possible to placing N queens on a $N \times N$ chessboard given the usual rules of freedom of movement of a chess queen. For the purposes of

# Clusters /Sim. Mode	Response time *Sim. Cycles*	Avg. Idle	Avg. Queue	Speedup	Efficiency
1/S	362986	0	1.565	NA	NA
3/H	250088	0.515	0.4839	1.45	0.48
3/T	18167848	0.992	0.006	0.02	0.0006
2/U	292490	0.4185	0.62	1.24	0.6205
12/Z	34985920	0.999	0.0007	0.013	0.00008

Table 5: Performance of Nqueens.

4.6 Summary

In this chapter a design for a distributed runtime environment for the efficient execution of a fine-grained object oriented language has been presented. This work contributes,

- The design of a distributed runtime for a fine grained active object model
- A infrastructure for efficient and flexible coordination of real and pseudo concurrency among components.
- A device independent communication strategy.

this benchmark, we have chosen $N = 7$. This results in 30 solutions to the problem, i.e., there are 30 different ways of placing 7 queens on a 7x7 chess board in a valid manner.

Most of the communication is synchronous as can be seen in Table 4.6. The simulation modes denoted in the table are: S for single cluster; H for our hints; T is a random clustering; U corresponds to results using [CR92]; and Z is one object per cluster. The most efficient performance is for the case of clustering by heuristic means although the response is not as good as for the case using our clustering strategies. The difference between the two is the one additional cluster required in the former. Availability of resources and the criticality of response time will dictate the exact clustering to be employed. The higher average idle time in our case, can be brought down by interleaving other applications onto the same processor. We have assumed dedicated processors which is not the case in a time shared environment. Clearly the Z mode yields catastrophic results due to the number of synchronous calls and their relative complexity. The random clustering behaves similarly in this case. The single cluster mode has numbers in the same order of magnitude but the loss of concurrency is clearly seen in the lower response time.

Chapter 5

Run-time Architecture

The Diamonds distributed runtime builds on and extends many of the structures of the basic runtime environment presented in Chapter 3. At the same time a new set of capabilities, such as remote object creation and invocation, are added.

In this chapter we will first examine the basic components that serve as the central entities in the runtime. Following this, we will then look further at some of the other abstractions and protocols that help organize the distributed runtime.

5.1 Basic Abstractions

The Diamonds runtime system is centered around 4 basic abstractions which may be defined as follows:

Application An application is the goal of the software development process as described in the requirements document. This is an intentionally vague definition, intended to model both large and small programs as well as interacting sets of programs.

Ensemble An ensemble is a set of clusters. An ensemble is an application's presence at a node. An ensemble is a pragmatic entity allowing a subset of an application's clusters to be mapped to a node.

Cluster A cluster is a set of objects. A cluster is a unit of performance optimization. The performance improvement can come from locality of reference, fast-path invocation or other means.

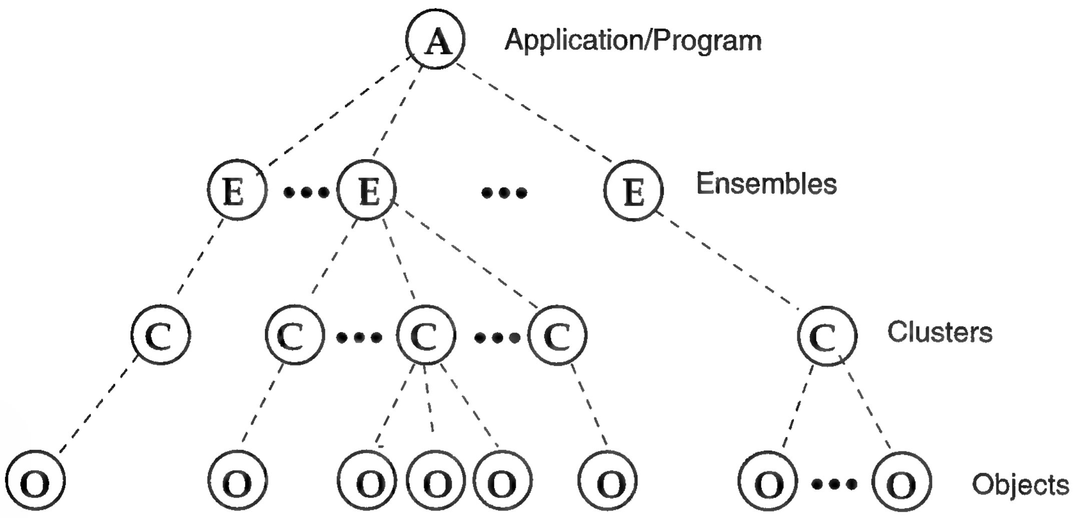

Figure 15: The Distributed Application Tree

Object An object is an instance of a class, which is a programmer defined software abstraction.

The hierarchical relationship among applications, ensembles, clusters and objects is shown in figure 15. We call this structure the *distributed application tree.* It is always rooted on the application node. The number of nodes at the lower levels of the tree will vary from one application to the next, as well the number of outgoing arcs from a node to its lower level nodes. The amount of variation will reflect both the nature of the application program and the dynamic resource state as an application executes.

5.1.1 Organization and Responsibilities

We reify the top three levels of figure 15 with corresponding software structures: `Program`, `Ensemble`, and `Cluster`. These Diamonds entities all cooperate in order to carry out the computations and communications on behalf of the program being executed.

`Program`

At the root of the hierarchy is the `Program` [1]. The `Program` is responsible for creating ensembles, and also serves as the coordinator for intra-Diamonds activities. It is also charged with the task

[1] `Program` and Application are used synonymously. The only distinction being that the first refers to the Diamonds software entity and the second, the abstract representation of programming task.

of determining when the application has completed execution and sending termination notices to all of the Ensembles.

The Program maintains information about the current location of all of the ensembles and clusters within the system

In our prototype design, the Program is a single software entity that resides at some node. In the final version of Diamonds, the Program would be a distributed entity so as to better tolerate system faults and failures.

Ensemble

An Ensemble is a manager of clusters. In the prototype design it is assigned to a Unix process or Mach task. It is responsible for creating its clusters and for forwarding messages from its internal clusters on to external clusters.

In order to assist in the sending of messages, the Ensemble maintains a table of ClusterRefs. ClusterRefs are mappings from cluster names to communications channels. Naming, references and communications will be covered more in subsequent sections.

If an Ensemble needs to forward a message to a cluster for which it has no information in its table of ClusterRefs, it sends a message to the the Program asking for information about the cluster. While this message is being sent and processed by the Program, the Ensemble will store the original message and forward it when the Program has replied.

Cluster

A Cluster is a manager of Objects. It is also an instance of a Supervisor class that was discussed previous in section 3.3.4. As an instance of a Supervisor, the Cluster becomes our unit of activity at the language level. The Cluster is responsible for *simulating* the activity of an object within our model. As long as an outside observer, or user of an odl program, cannot tell whether objects are actually active or simulated, we place no restrictions on how the activity is achieved.

Since the semantics of some language level operation such as New change from the non-Diamonds environment of the Supervisor to the environment of the Cluster, those operations are implemented via virtual function calls (i.e. upcalls).

A Cluster acts as an odl objects' intermediary to the outside world. In this capacity it must translate messages that are generated at the language level to those that are understood at the Diamonds level and back again. To assist in this operation, each Cluster has a MsgTranslator whose only task is the translation (in either direction) between odl and Diamonds messages.

5.1.2 Intra-runtime Protocols

One of the biggest challenges in implementing Diamonds was simulating and controlling the large amount of concurrency (both real and pseudo) that is at the core of the whole programming model of Diamonds. This is a challenge because the machines on which Diamonds was prototyped support only large grained activities (i.e. a Unix process) and not the finer grained units of activity (`Objects`, `Clusters`, `Ensembles`, and `Programs`) that are present within Diamonds.

To illustrate this challenge, consider the distributed application tree that we pictured in figure 15. Each node in this graph is potentially a separate, concurrent activity. In mapping Diamonds to a workstation environment, there is true concurrency at the top two levels, Application and Ensemble. There is also true concurrency between Clusters of different Ensembles. Between Clusters of the same Ensemble we have only pseudo-concurrency since they must ultimately share the same CPU.

This is just one possible mapping. As a design goal we desired to support nearly *all possible* mappings, ranging from true concurrency among all nodes of the distributed application tree (where each node in the graph might be assigned a CPU within a multiprocessor or set of multiprocessors), to only pseudo concurrency among all the nodes (where all elements of the tree might execute within a single unix process).

In order to achieve the level of true and pseudo concurrency, and to use the same body of code in all cases, we designed a set of intra-runtime communications protocols. The heart of this design is a message passing infrastructure for our key Diamonds components, `Program`, `Ensemble` and `Cluster`, and a set of location independent naming and communication primitives. In the next two subsections we will examine this design as it reflects on our elements of the distributed application tree, and then in subsequent sections discuss the communications and naming designs.

Agents

The intra-runtime protocol specification is encapsulated within what we call an `Agent`. Each of our key software entities, `Program`, `Ensemble` and `Cluster` are is an instance of `class Agent`. This relationship is depicted in figure 16. An `Agent` has a `MsgPort` on which it accepts messages from other `Agents`, and a `MsgQueue` where it stores messages between acceptance and processing. A more detailed representation of `Agents` is presented in figure 17. The details of message delivery, acceptance, etc., are completely hidden from the `Agent` and encapsulated by the `MsgPort`. `MsgPorts` are covered in section 5.2.2.

Message Handling

The Diamonds agents all implement the protocol for message passing which is specified by the abstract `Agent` class. All of the internal operations of `Agents` are dependent on messages,

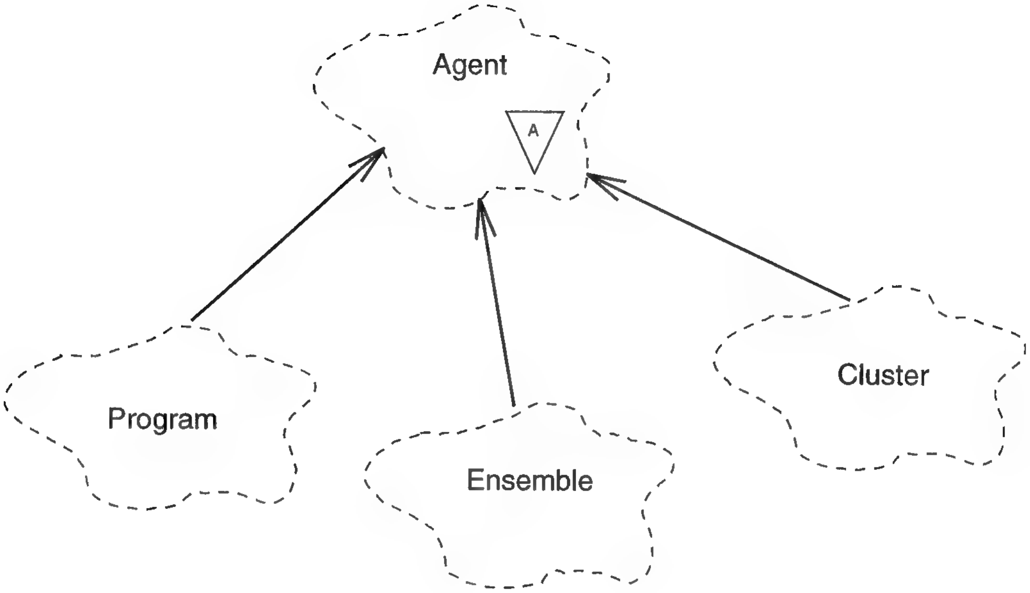

Figure 16: Diamonds Agents

meaning that communication from one agent to another is done via messages.

Each `Agent` is able to handle a different set of messages and define its own semantics for what is done when a message of a certain type is received. The messages understood and semantics implied by each are presented in table 6 for `Program`, table 7 for `Ensemble` and table 8 , for the `Cluster` agents respectively.

True & Pseudo Concurrency

One of the major challenges in creating an environment for a fine-grained active object language is the management of both true and pseudo concurrency. In any odl program we have the potential for both types. There is true concurrency when we are able to utilize two or more CPUs for the same application, and clusters are active simultaneously on the multiple CPUs. There is pseudo concurrency in any application when multiple clusters reside in a single ensemble, and that ensemble is mapped to a single CPU. There can also be pseudo concurrency within a cluster itself when multiple objects have messages sent to them.

Ultimately, if we are going to run an odl application on today's hardware, we will have to come up with some scheme for accurately and efficiently simulating the active object model. This is because we are unable to assign either a CPU or a "process" to every object. It is also likely that due to the resource waste that would occur if we were able to do this, we would not

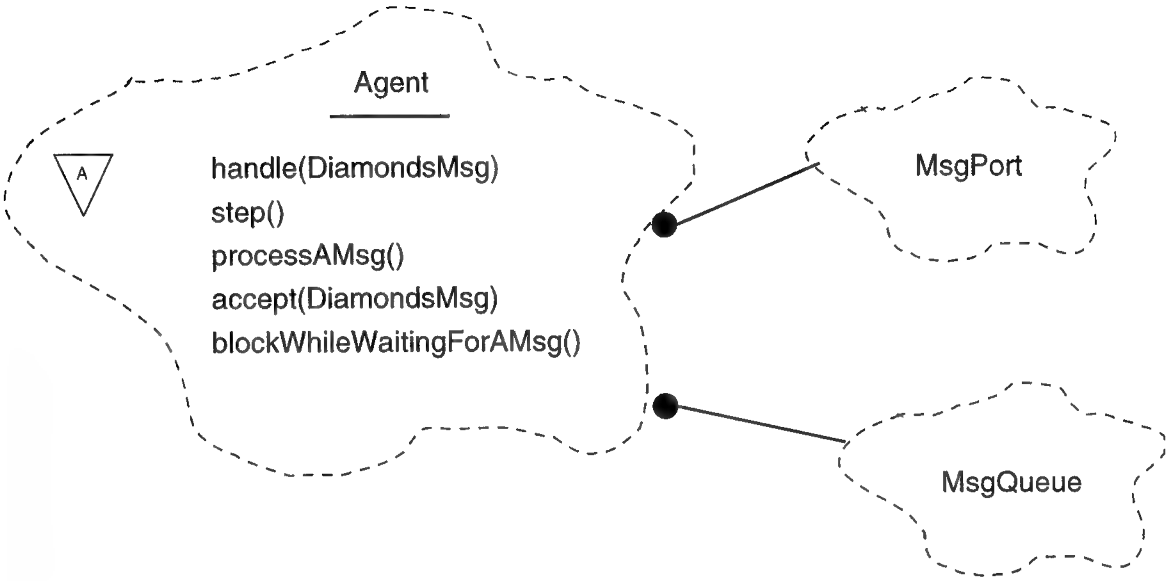

Figure 17: Agent Design

want to.

What we have done within the Diamonds prototype is to design a thread sharing protocol among `Agents`. Our use of the term *thread* here is in its most abstract definition, as nothing more than a thread of control. Whether there is an actual operating system thread abstraction is orthogonal to the design discussed here.

Our thread sharing protocol takes advantage of the hierarchical relationship between the agents `Program`, `Ensemble` and `Cluster`. For each `Agent`, we specify a *unit of work* to be,

$$unitOfWork == processAMsg + handleAMsg + shareThread$$

where the distinction between processing a message and handling it is one of acceptance and removal from the buffer while the later is a scheduling decision.

For thread sharing, we have constructed a flexible "subAgent" arrangement. Any `Agent` can register itself as a *subAgent* of another. The logical relationship between the two in this case is that the *subAgent* falls below the `Agent` in our distributed application tree, and is also sharing an address or process space with the `Agent`. For most mappings of the Diamonds runtime, `Clusters` will register as *subAgents* of their `Ensemble`. In the situation where the entire runtime is to share

Message	Semantics
`NextClusterId`	Respond to sender with a new unique id for a cluster
`NewClusterInfo`	Sender is supplying location and communications information about a cluster. The cluster may be new or one that has recently migrated.
`WhereIsCluster`	Sender requests location information about a certain cluster.
`NumMsgQueued`	Sender is supplying information about the number of messages queued and available for processing.

Table 6: Program Messages

a single process [2], each `Ensemble` will register itself as a *subAgent* of the `Program`.

In this *subAgent* model, a *unitOfWork* begins at the `Program` level. A `Program` can *processAMsg*, if any, and then allow each of its registered *subAgents* (`Ensembles`) to perform a *unitOfWork*. This sharing of the "thread of control" will then continue down the application tree as `Ensembles` perform a *unitOfWork* and then allow their *subAgents*(`Clusters`) to perform a *unitOfWork*.

The definition of a *unitOfWork* differs slightly when we reach the leaves in our distributed application tree (the `Clusters`). Here, there are no *subAgents*, and the work that must be done is to progress with the language level computation. So, for a `Cluster`, a *unitOfWork* is defined as,

$$unitOfWork == processAMsg + handleAMsg + runAContext$$

The *processAMsg* and *handleAMsg* are as before, but now, instead of *sharingAThread*, the `Cluster` executes a `Context`. This `Context` may have been created during this *unitOfWork* or may be one from a previous cycle. The exact semantics of *runAContext* are covered in section 5.6.4 where scheduling is discussed.

It is through this *subAgent* control flow protocol and our message passing interface that we are able to realize both true and pseudo concurrency for a fine-grained object model on common workstation architectures.

[2] This mapping of all `Agents` to a single process can be extremely valuable for debugging purposes.

Message	Semantics
Initialize	Sender (the Program) is supplying information the SIRF file to be run and the number of local clusters
ClusterLocation	Accept information about the location of a cluster
ExecuteMain	Look up and begin executing the odl method. System::main
Terminate	Quit processing messages and exit.
SetMsgCounter	Reset the message counter to allow for processing of at most N messages
Step	Execute a single message (if possible)

Table 7: Ensemble Messages

5.2 Communications

In this section, we will discuss the design of the components on which messages are sent over the distributed system. In section 5.4, we will discuss how these components are integrated into Diamonds.

The prototype set of communications utilities are built on components which abstract BSD Sockets [LFJ+86]. These components are designed to hide any details about the actual communications facilities (i.e. Sockets) so that other types of communications could be added later (i.e. Mach IPC [Dra90]).

The communications abstractions are based on an abstract base class, ipcJunction which specifies the interface that all subclasses must support. In this case, it is basically the ability to get or send a byte string and to answer queries about its status.

As special types of ipcJunctions, we have the Socket class. The semantic differences in the use of sockets (i.e. stream vs. datagram) are captured by subclasses of Socket: ipcStream and ipcDatagram. A pictorial representation of these relationships is given in figure 18.

5.2.1 Communicators

One of the primary goals in the design of the runtime system for Diamonds was the support for a flexible topology of Agents. Diamonds is targeted towards an environment where resources change and fluctuate. This precludes the ability to say “Ensembles always get a separate node”, if we only have 1 node up, and we still want to run our odl program and get some results.

Message	Semantics
ODLMessage	Body of message is a language level method invocation.
Reply	Callback from a language level method invocation.
CreateObject	Create an object within this cluster on behalf of some object.
ObjectCreated	Callback message for a remote "CreateObject" message which was initiated by an object within this cluster.

Table 8: Cluster Messages

What this implies for the designer and implementor of such a system is that we cannot make assumptions about the relative location of one Agent to another, or the type of medium by which they communicate. We have done this by using a naming convention that does not bind an Agent to a node, and a communication protocol that does not reveal to the sender the type of medium that is used to transport the message. We will discuss the details of our communications protocol now and defer the discussion of naming until section 5.3.

We base our communication protocol on a Communicator object. It is a unidirectional means of transmitting a message. The most common use of Communicators within Diamonds is for sending a message from one Agent to another. Messages are always received by an Agent at its MsgPort. The Communicator and its subtypes are shown in figure 19. With this design a Communicator (the abstract base type) can be used throughout the program without any knowledge as to whether the actual type is local or remote, and if it is remote, what the semantics of the remote communications are. This allows us to freely substitute a Communicator of the appropriate type, based upon the dynamic, relative location of the Agents. An alternative to this sort of architecture would be one where the locations of the sender and receiver are evaluated each and every time a message is sent, as sketched in pseudo-code in figure 20. The drawback to this approach is that that must perform this check each and every time we send a message. Depending on the design, this check might take place at many points in the code. Adding an additional type of location specifier or a new transmission protocol requires that all of these sections of code be found and modified. This is a classic example of the benefits of object oriented inheritance and the transparency it allows in client code with regards to the actual handling of operations.

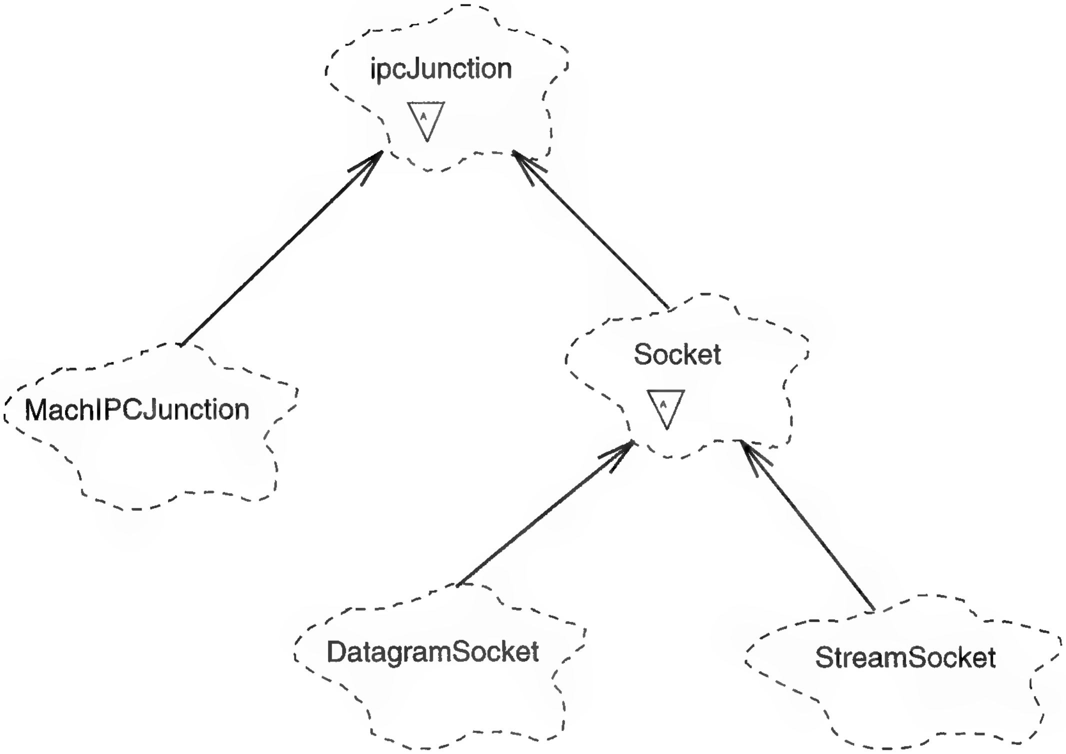

Figure 18: Subtypes of ipcJunction

LocalCommunicators as an optimization

A LocalCommunicator is a means of directly accessing the MsgPort of another Agent in a structured and protected manner. Within a LocalCommunicator is the address of the MsgPort of the Agent to which it is communicating. Message sends through a LocalCommunicator can then directly add the message to the MsgPort buffer, without incurring the overhead of an operating system context switch.

One way of viewing LocalCommunicators is as an optimization for the case where Agents are sharing an address space. Note that the RemoteCommunicator protocol would still be possible, but would add the overhead of kernel context switches and data passing for what is actually possible at the user level. LocalCommunicators remove this kernel overhead in a clean and transparent manner.

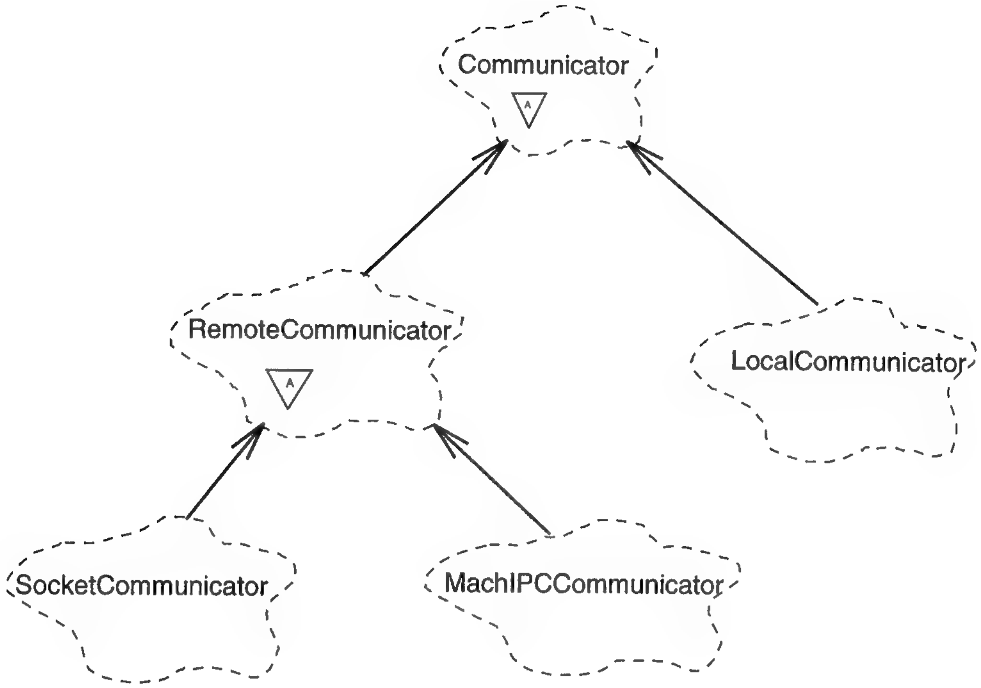

Figure 19: Diamonds Communicators

5.2.2 MsgPorts

A MsgPort is an acceptance point for messages. Each Agent has a message port to which other Agents may send messages. The means by which messages arrive at the message port are abstracted from the MsgPort itself, as is shown in figure 21. The communications medium by which a MsgPort receives messages is a parameter that is established at the time a MsgPort is created. The implication of this design decision is that once it is created, a MsgPort has a "hidden" communications type that is fixed for its lifetime. In Diamonds, this is not really an issue, since the type of ipcJunction for a MsgPort should be the most generic one possible, based on the network interface. This is often a Socket type abstraction since in our heterogeneous environment of both machine types and operating systems, BSD Sockets are almost universally understood.

When an Agent migrates, the MsgPort structure itself will not migrate with it. It really cannot, since the implementation must be redefined within the context of the Agent's new environment. For example, what meaning does a MachIPCJunction have in a pure BSD environment? The migration protocol must deal with the re-establishment of proper peer-to-peer protocols following the migration. The naming and reference structures, which will be discussed

```
 // If Agent X wants to send message A to Agent Y
 // X must perform the following

if (Y.location() == X.location())
   sendUsingLocalAddressSpace(A,Y);
else if (Y.location() == X.node())
   sendUsingLightWeightProcedureCall(A,Y)
else
   // note --> Y.location() != X.node()
   sendUsingFullNetworkProtocal(A,Y)
```

Figure 20: An alternative to Communicators

in sections 5.3 and 5.4 respectively, have been designed with this need in mind.

5.3 Naming

Each of the four basic abstractions within Diamonds must be named if it is to be the destination of a message.

Naming within distributed systems has long been, and continues to be, a topic of research. We have chosen a naming scheme that we believe matches our primary needs of extensibility and efficient execution.

One of the most basic decisions in a naming scheme is whether a name also encodes location information. The most familiar naming scheme to most people, pointers, is an example of encoding location and naming information. In our environment, static location information is non-existent, and dynamic location information can change since clusters can migrate. For these reasons, we have chosen to identify Agents and Objects within Diamonds in a manner that does not include explicit location information. As we shall see, our naming scheme does implicitly encode some *relative* location information.

We have two levels of naming to be concerned with. This first and most important is language level naming. How do the language level objects identify one another? The second level is intra-Diamonds and is concerned with the question, "How do the Agents identify one another?".

We will first look at each of the Agents within Diamonds and then at the language level

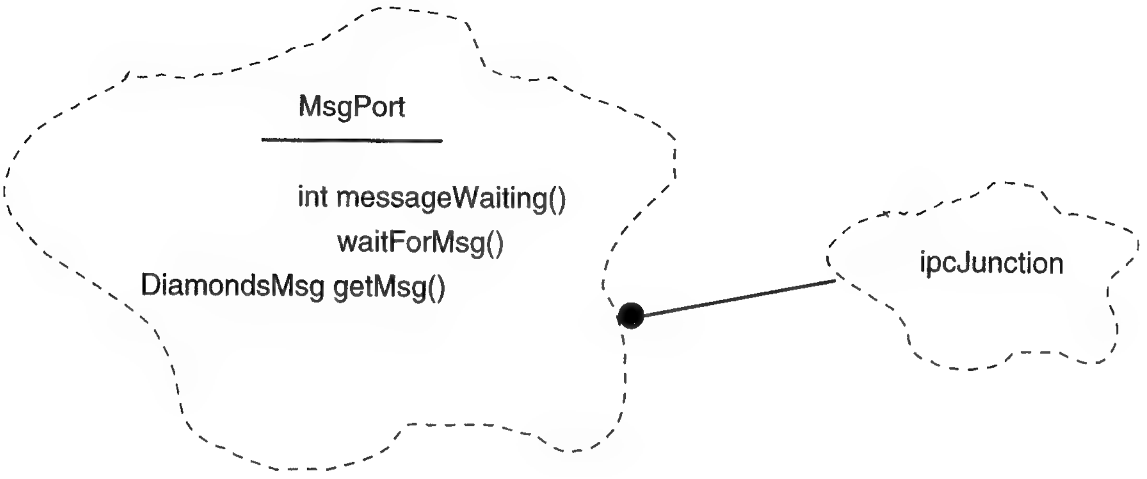

Figure 21: `MsgPort` design

issues.

`Program` There is, at the top level of the distributed application tree within Diamonds, a `Program` entity. In the prototype, the `Program` is the intermediary between the user and the code, performing such operations as I/O and event signaling. Both conceptually with the Diamonds model and actually with the prototype, there is a single `Program` entity. For this reason we are able to *name* it completely by its communication port on which it receives messages.

`Ensemble` An `Ensemble` is named with an unique integer issued by the `Program`. Having a single entity issue the numbers helps to insure that the numbers are unique. Given the scale of Diamonds, the integer identifiers for `Ensembles` will typically be in the range of 1 to 100. An `EnsembleIdentifier` abstracts this underlying implementation for the rest of the program.

`Cluster` A `Cluster` is initially created by an `Ensemble`, but given the ability to migrate, there is no long term association between the creating `Ensemble` and the `Cluster` it has created. This observation tends to disfavor a relative naming scheme involving ensembles and clusters. Instead, we have chosen to identify `Clusters` in a manner similar to `Ensembles`, with a unique integer. These identifiers are allocated by the `Program`, so an `Ensemble` must request a new `ClusterIdentifier` from the `Program` for every `Cluster` it creates. The range of the integers identifying clusters might be an order of magnitude greater than those describing ensembles.

In selecting an identification/naming scheme for a language level object, we must first examine how objects are created, managed and accessed. In Diamonds we give the entire responsibility for object management to the Cluster. It does the actual creation and also accepts messages destined for an object on that objects behalf. In our current design of Diamonds, we have ruled out cluster fragmentation [3]. This means that an object will always be associated with the same cluster throughout its lifetime. For this reason, we have chosen a relative naming scheme for Objects where their name is a combination of their cluster's ClusterIdentifier, and a unique number. The unique number is again an integer. It is allocated and assigned by the cluster.

The advantage of this relative naming scheme for objects is that object creation need not consult a global name table to select a free entry or consult with other Agents to ensure that a new name does not clash with an existing one. Since the name of a cluster is unique within the system, and objects within a cluster are uniquely identified, we are guaranteed that the combination of the ClusterIdentifier and ObjectNumber will produce a ObjectIdentifier that is globally unique with respect to other objects within this application.

If two or more applications are going to cooperate, either by simply exchanging messages or by "sharing" a cluster, an additional level of naming and access control will be needed. One means of mediating access to intra-application objects would be through a name server where externally applicable names could be registered and associated with intra-application ones. This would avoid the potential clash of Object and Cluster names between applications.

5.4 References

In the last section on naming, we stated that names or identifiers within Diamonds do not include location information. Given a name, what we almost invariably want to do with that name is to communicate with the object to which it refers. So, within a system like Diamonds, the association of names with communications is of more use than an association of names and location. We formalize this with a set of abstractions which we refer to as *references*. A reference is a name and a communications medium with the named object. The actual type of the communications medium is not visible to the user of the reference. This is an important part of our design and one of our primary means of achieving the flexibility that we are able to within Diamonds.

The reference structures within Diamonds are pictured in figure 22. The base type Reference includes a link to a Communicator as well as a method to send a DiamondsMsg to the object referenced. The subtypes of Reference add type specific naming information. So for instance, a ClusterRef contains the ClusterIdentier of the cluster it references.

With this design we are able to distinguish identity, since two references can refer to the

[3] Cluster fragmentation is the dynamic operation of splitting a single cluster into two or more new ones.

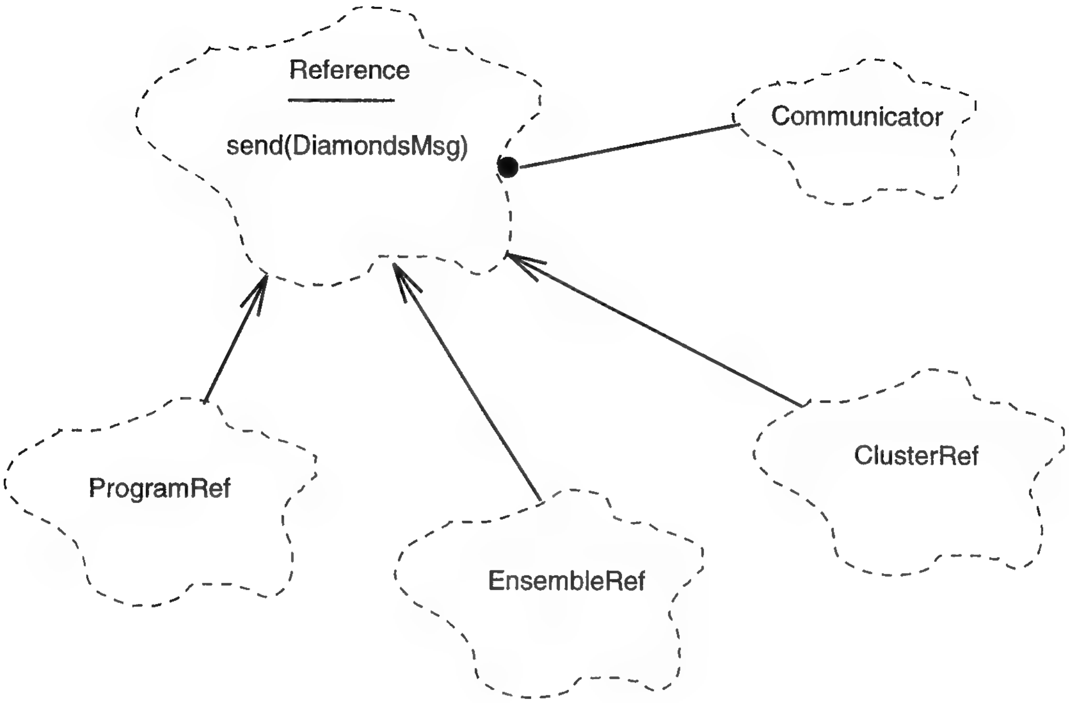

Figure 22: Reference Structure

same object if and only if they are first of the same type and then secondly if their identifiers are the same. We are also able to send messages to an object via the `Communicator`. We are able to both of these tasks without knowing the location of the object which is being referenced. The only time the location information is necessary is when the `Reference` is first made, and later in response to cluster migration or node failures. Either way, the *users* of a `Reference` are not aware of the location or changes in the location.

The location independent referencing structure that we have discussed is a significant contribution of Diamonds. It is this structure, in part, that allows us to easily and transparently execute the same SIRF program on a single node or hundreds of node. This is made possible in conjunction with the `Communicator` structure that we have previously discussed(see section 5.2.1). When a `Reference` is created we can select the type of `Communicator` which it will use. If the referenced object is in the same address space as the referencer, we can use a `LocalCommunicator`. If the referenced object is remote, we can select from among the remote communicators. Either way, the choice is not visible to clients of the `Communicator`. This allows us to get both the efficiency of `LocalCommunicators` where they are applicable, and to treat `References` as a single consistent entity.

The references that we have discussed so far are all inter-Agent type references, used by the runtime to uniformly communicate with Agents. There is another reference structure that is not related to the references we have discussed so far. That is an ObjectRef. It is used at the language level to communicate with language level objects. It helps to abstract the location of an object as do the other references.

5.4.1 Optimizations

A criticism of the naming and reference schemes as presented in sections 5.3 and 5.4 concerns the naming of language level objects. So far we have said that objects are assigned a unique integer and then named relative to their cluster. In any significant program there might be *millions* or *billions* of objects created and destroyed in its lifetime. The overhead for creating names and managing names to all of these objects would be prohibitive.

We recognize this problem in our design and have dealt with is by making the observation that a great many objects will *never* be externally referenced. That is, all access to the object will be done by other objects in the same cluster as it. In this case the whole ObjectIdentifier, ObjectRef protocol is really overkill, and nothing but unnecessary overhead.

In order to avoid this overhead we need to introduce a *pre-clustering* phase prior to the application of the static clustering algorithm presented in Chapter 4. We call this phase, *attribute subsumption*. The purpose of this phase is to identify simple clusters of objects whose identities are never exported and whose access is entirely procedural. This will be the majority of the objects in our system, encompassing all of the most familiar "builtin" types such as ints and chars. For these objects we can simply patch in a call to a SIRF new instruction [4] that creates the object and returns a pointer to it directly. For all of the other calls to new which may generate an object which will be externally referenced we will patch in a version of new which does create the full ObjectRef [5]. The details of attribute subsumption for object oriented languages in general and its application within Diamondsare discussed in section [Sha94].

In conjunction with *pre-clustering* the same substitution of optimized naming structures could be used as a *post-clustering* process. In this phase the results of static clustering can be used to additional sites where anonymous naming may be used.

5.5 Initialization

There is quite a bit of pre-program-execution activity that must go on within Diamonds in order to properly establish the runtime environment for execution. In this section we will look at some

[4] This is instruction is called LiteNew.

[5] This new instruction is called HeavyNew.

of the activities that occur within Diamonds prior to the execution of the first SIRF instruction.

The basic order of construction of the Diamonds Agents is from the root down, as pictured in figure 15. Each node at one level is responsible for creating its immediate sub-nodes.

5.5.1 Program

The Program is created first, and it establishes its MsgPort on which it will receive requests from other Agents. The first task of the Program is to create Ensembles for the program to execute. The number of Ensembles is both a characteristic and a compromise. It is a characteristic of the program as determined by static analysis. It is a measure of the maximum potential parallelism with the program. It is a compromise too because no matter what the *desired* number of Ensembles, the actual number used will reflect the dynamic number of machines available and the current state of the system wide resources.

In the prototype implementation of Diamonds the Program startup code is designed to initially interact with a resource manager who will decide how many Ensembles to use for the program.

As part of the startup process the Program starts new Ensembles. A new Ensemble may be created in one of several ways,

- Automatically, as a local process on the same machine. The program will just do a fork() and exec() with an identifier for its MsgPort passed to the new process.
- Automatically, remotely. This can be done via a rsh command, again with an MsgPort identifier as an argument.
- Manually. The developer can manually start up an ensembles if that option was specified to the program when it was run. (This is just a debugging option and not expected to be used by the casual or end user of Diamonds).

After a Ensemble is started the Program enters into a start-up synchronization protocol sequence with the new Ensemble in which communications and other information is exchanged. After the protocol sequence the Ensemble is an independent Agent that carries on as we describe below.

Once all of the Ensembles have been created, some Ensemble must be chosen to begin interpretation of the SIRF program at what corresponds to the main() method from within the odl code. In the prototype implementation the first Ensemble created is always selected as the one that executes main().

5.5.2 Ensembles

A Ensemble is responsible for creating a certain number of Clusters. This creation process could be demand driven, meaning that they are created as the program executes, or apriori.

In the prototype implementation of Diamonds Ensembles create Clusters in an apriori manner. This process begins after they receive the *initialize* message as part of their startup synchronization sequence with the Program.

As we mentioned in section 5.3, cluster names are unique within the application. The Agent responsible for issuing cluster names is the Program. When an Ensemble wishes to create a new Cluster it first sends a message to the the Program, *nextClusterId*, requesting a new ClusterIdentifier be allocated for the Ensemble. The Ensemble blocks until the clusterIdentifier is received. Once it has received a ClusterIdentifier it creates the new Cluster. This process is carried out for all of the clusters the Ensemble is to create.

5.5.3 Clusters

The set-up and initialization is for Clusters is much simpler than that for Ensembles since Clusters are created within the same address space of the Ensemble. A Cluster just initializes its attributes and goes into a wait state, waiting for messages to be sent to it.

5.6 Execution Protocols

The Agents must cooperate in order to handle actions which are carried out by two or more Agents. In this section we will look at some of the more interesting inter-Agent tasks within Diamonds.

5.6.1 Object Creation

There are two possible types of new objects with respect to a cluster. The first type is *local*, in which case the cluster whose object is executing the SIRF new statement is also the cluster in which the object is to be created. The second type is remote, where another cluster is chosen as the site for the new object. The decision as to where an object is created is completely independent of the mechanics of local or remote creation.

If an object is to be created locally then the cluster can directly carry out the new and allow the execution object to continue with a handle to the object that has been created.

In the case of a remote new the originating cluster (the cluster executing the SIRF new request) must send a message to the generating cluster (the cluster chosen to actually create the new object). This message identifies the class type of the object to be created and the ClusterId of the originating cluster. This message also includes a *Key* which uniquely identifies this request from others that might be generated by this cluster. The *Key* is necessary because the originating cluster may send remote new messages to many other generating clusters on behalf of its objects, some of these requests might also be for the same class type. The *Key* allows one request to be completely disambiguated from all of the others.

The *createObject* message is forwarded to generating cluster by the ensemble of the originating cluster. The request to the generating cluster is treated as a synchronous message send with respect to the method currently being executed. The `Context` that is currently executing must be suspended until the generating cluster completes the creation and reports this fact to the originating cluster. In order to handle this the `WaitQ` facility is used(see section 5.8 for details on `WaitQs`). A `RemoteNewWait` object is created, associated with the active `Context`, the `Key` of the message that has been sent, and an new empty `ObjectRef`. The new `RemoteNewWait` object is then queued, effectively suspending the current context until receipt of information from the generating cluster.

To preserve the semantics of local object creation, the empty `ObjectRef` is returned by the cluster to the `VirtualMachine/ExecutionAgent`. This allows for the addition of a `Future` style interaction [ATK92] to be added later if desired. At this point the the `Cluster` specifically requests the `ExecutionAgent` to reschedule, forcing the selection of a new context to be run.

The `Context` of the the originating object is suspended until an *ObjectCreated* message is received from the generating cluster. The `Key` of this message must match the key of the *CreateObject* message which was originally sent. If there is a match, the data within the *ObjectCreated* message is used to fill in the `ObjectRef` which was created earlier and stored within the `RemoteNewWait`. The `handle()` method of the `RemoteNewWait` extracts information from the *ObjectCreated* message and build a new `ObjectIdentifier` which is used to fill in the `ObjectRef`. At this point the suspended context can be added to the runnable queue, effectively completing the synchronous remote object creation.

5.6.2 Object Invocation

When a method or operation invocation is made from one object to another the cluster of the calling object intercedes to handle the dispatch. The cluster first checks to see the *type* of object handle on which the operation is being invoked. If it is a local anonymous or named object, the cluster contains both the client and server object in the interaction. In this case the cluster can directly create a new `Context` based on information it can lookup about the type of method being called, its parameters and the number of local objects within it. This new `Context` is queued with its status set as *runnable*.

If the handle to the object being called (the server) is an `ObjectRef` (an object reference) then the server is in some other cluster. In this case the cluster of the client first builds a `DiamondsMsg`. The specific type of the `DiamondsMsg` is `ODLMessage` so it includes all of the parameters to the method being called as well as information about the object and context (see section 5.6.3) generating the call. The ObjectRef identifies the cluster of the server object and this information is then passed to the client cluster's ensemble who forwards the message on to the server cluster.

At this point the calling cluster checks the type of method being called. If it is asynchronous then the current object can continue to execute. If it is synchronous then a new ReplyWait object is created. The ReplyWait is associated with the Key of the DiamondsMsg which was sent, as well as the current context. This ReplyWait is then queued in the WaitQ, suspending the current context until a *Reply* message is received from the server object.

From a language level perspective, there is no difference from a remote or local invocation. None of the details of the remote invocation protocol are visible to the programmer or user of the program.

5.6.3 Callbacks

We use callbacks instead of other distributed invocation methods such as RPC [Bir85, ATK92] to carry out synchronous remote invocations.

The implementation of callbacks in Diamonds is a little bit different than one might expect. This is because we are forced to deal with the results of some optimizations we have made elsewhere. As we discussed in section 5.3, not all objects receive full blown names. We have many anonymous objects because their identity is never exported beyond a local clustered set of other objects, and thus cannot be targets of a method call from an object in another cluster. This however does not preclude them from making an invocation on an object within another cluster. The combination of these two issues is what forces our implementation of callbacks to be a bit different than what one might expect.

One very "clean" method of implementing callbacks would be to make a callback look as close to a method call of any other type as possible. This would imply that there is an ObjectIdentifier associated with the object that is being called. As we said in the previous paragraph, a local anonymous object, i.e. one *without* a ObjectIdentifier, can make an invocation on a remote object. In this case there is no ObjectIdentifier to associate with the callback if the regular method call semantics are to be used.

In order to handle this we have a several options. This first would be to statically analyze the program and attempt predict which objects will be making remote invocations, and give those objects an ObjectIdentifier, making them remotely addressable. Since the dynamic placement of objects is unknown staticly we would be left with no option but to name nearly all objects with ObjectIdentifiers, defeating the whole purpose of the optimizations allowed by the dual naming scheme that we discussed in section 5.3.

A second option would be to dynamically "promote" the name of an object from anonymous to named, creating an ObjectIdentifier for it and allowing the call and subsequent call back to proceed. The problem here is that this new ObjectIdentifier introduces a local alias for the anonymous object. In subsequent sections of code we will be unable to determine the equality between to names to the same object if one is of type "local anonymous" and the other is of type "ObjectIdentifier". This is because we use pointers as "local anonymous" names and have no means to dynamically find all reference to the anonymous object and replace them with the new `ObjectIdentifier`.

The solution to this callback problem that we have adopted is to name the `Context` of the object making the remote invocation. The name of this *context* instead of the name of the *object* is used when the remote object makes its callback. This allows local anonymous objects to make remote invocations, and allows us to continue to use our lightweight dual naming scheme.

Contexts are named in a manner similar to objects. They are numbered monotonically and then named via a "clusterNumber" "contextNumber" pair.

A callback is made in response to a `Reply` SIRF/odl instruction [6]. A *Reply* type `DiamondsMsg` is generated which contains the return value (if any) and this message is forwarded on via the local ensemble to the calling object.

When the *Reply* message is received by the calling cluster, the `Key` value within the message is used to select the `ReplyWait` object from within the `WaitQ`. The `handle()` method of the `ReplyWait` takes care of extracting the return value from the message and storing it in the saved reply register within the context. The context is then set as runnable and returned to the active queue.

5.6.4 Scheduling

Diamonds allows for many different scheduling policies to be used within `Clusters` to select the next `Context` to be executed. In the prototype we use a simple round-robin scheme. All `Context` are associated with the same priority. This is effectively realized by using a single "active" context queue. Contexts are selected from the queue in a FIFO manner. A context is allowed to execute until it generates a message. Given the fine-granularity of message sends within odl code this is an acceptable scheduling point for the present time. When a context generates a message it is placed at the end of the queue. When pended contexts become runnable (i.e. a *guarded* condition becomes true, or a synchronous remote event completes) the context is placed at the end of the queue.

5.6.5 Unknown Clusters

In our messaging scheme a reference to a `Cluster` can be generated prior to there being any location or communication information about the `Cluster`. This can occur when the first `new`

[6] The odl compiler silently inserts a `reply` instruction which returns no value at the end of any synchronous odl method which does not explicitly contain a `reply`.

is encountered for an object in a new Cluster. This can also happen when a ObjectRef is passed as a parameter to a Cluster/Ensemble pair which has not previously sent a message to the Cluster of the referenced object.

In either case it is the responsibility of the Ensemble to forward messages from its Clusters. If an internal Cluster generates a message to an external Cluster for which the Ensemble lacks a valid or complete ClusterRef, the message cannot be sent. In the case where it lacks a ClusterRef for an external Cluster the Ensemble will send a *WhereIsCluster* message to the Program. The Ensemble is unable to forward the message to the external Cluster until it receives a reply to its *WhereIsCluster* message. In the mean time it appends the cluster's message to its StoreAndForwardQ, which is nothing but a temporary holding area for these sort of messages. At this point the cluster which created the message can resume execution, oblivious to the fact that its message may or may not of been sent at this time.

From a Cluster's perspective a message to a locally unknown Cluster is no different than any other. The actual difference is that the message delivery process is just delayed by the lookup time associated with the Ensemble to Program Cluster location message.

The Program contains a complete mapping of Clusters so the Ensemble is assured that it will receive an answer to its *WhereIsCluster* message. The reply to this inquiry is itself in the form a message, *ClusterLocation*. When a Ensemble receives a *ClusterLocation* message it builds a new ClusterRef structure with a Communicator of an appropriate type. Once this is completed it then checks its StoreAndForwardQ for any messages which are held waiting for information about this Cluster. If found, those messages are then forwarded on.

The description of the unknown clusters problem to this point has been in terms of straight forward message sends, in response to new references to as of yet unknown clusters. This same mechanism can also be used to facilitate the migration of *existing* clusters. In a possible implementation of this scenario the first step in the migration of a cluster would be to broadcast a request to all Ensembles to invalidate their ClusterRef for the cluster about to undergo migration. In response to this, new message generated at the cluster level will be queued and stored until the Ensembles receive updated information. This is the same interaction protocol that we described in the previous paragraphs. Once a new location for the migrating Cluster is established, this information would be registered with the Program, who could then respond to any enquires it might have had in the mean time as to the location of the Cluster. Once Ensembles receive the updated location information they will be able to forward on stored messages.

5.7 Runtime Bookkeeping

In coordinating computations in a distributed system some other facilities are needed beyond those that we have discussed to this point. Specifically, there is a need for both controlling and monitoring the progress made by each cluster and ensemble. This information can also be used to detect deadlock (in some cases), assist in application driven resource management and to determine when the application has completed.

5.7.1 Message Counts

As a bookkeeping and an initial step towards being able to run in a "debug" mode each Agent within Diamonds executes a certain number of "steps". We have previously discussed what it means to perform a *unitOfWork* at each level in the Agent hierarchy. We define a single step to be one *unitOfWork*. The Program controls how many steps an Ensemble or Cluster is able to perform before it stop and waits on further instructions from the Program. In a debug mode each Agent might be allowed to perform only a single step (one *unitOfWork* before stopping and waiting further information). In the prototype design the default number of steps that each Agent is allowed to perform before re-synchronizing with the Program is 5. The synchronization step between Agents is primarily used to coordinate termination in a distributed setting as we will discuss next.

5.7.2 Termination

When a group of independent process are all cooperating to perform a single computation it can be surprisingly difficult to figure out the global state of the computation. For the most part the exact global state need not be known. There is however one important state that must be detected and that is the final or termination state.

In order to do this within Diamonds we again make use of our hierarchical model and the synchronization sequences that occur as each Agent has completed their allotted number of "steps" or units of work. The basic protocol is as follows

- Each Ensemble reports the total number of messages queued within the Clusters that it manages to the Program at the end of each *work* cycle. A work cycle is determined by the number of "steps" the Ensemble is allowed to take before re-synchronizing with the Program. This can vary from 1 to nearly infinity.

- The Program keeps track of the number of messages queued on a per Ensemble basis. Reports are expected from each Ensemble at the end of every work cycle.

- The presence of queued messages in any Ensemble is an indication that there is work still to be done. If this is the case, the Program waits until reports have been received from all Ensembles. When this is completed, it then broadcasts a new "work cycle" message to the Ensembles. This work cycle message is in the form of the number of steps that they may perform before reporting back.

- If, at the completion of a single work cycle, where all Ensembles report that there are no message queued, it is likely that the computation or application has completed. The presence of delays in the network, however, makes it possible that a message in transit has not been accounted for. In order to handle this the Program waits for two consecutive work cycles with no messages queued in any Cluster before concluding the application has completed.
- If the Program determines that the application has completed it sends a termination message to all Ensembles who then can perform any necessary clean up operations before exiting.

5.8 Other Components

There are a few other abstractions that are used by the runtime to facilitate management of messages and Agents. In this section we will touch on the highlights of a few of these.

5.8.1 Pending Actions

In many situations a computation cannot proceed until another event has occurred. This can be signaled at the language level through the use of odl guards, or at the runtime level as a consequence of the enforcement of language level semantics (as is the case when a computation must be halted until a synchronous method call completes).

The actions to be taken after an event that must complete does, varies for each of these situations. They all share some commonality in that the currently executing context will not be a candidate for further execution until a particular message arrives. We manage this with a set of WaitCondition types that implement the exact behavior and manage all of them through a common interface and store them in a WaitQ(a queue of WaitConditions).

A WaitCondition is associated with a local Context and a DiamondsMsg key. The key is used to distinguish and route particular messages. In case of a reply from a remote method call, the key was generated locally, passed in the original message and returned as part of the reply. The key is what signals the system as to what WaitCondition should be selected to handle the message. So when a *keyed* message is received at the Cluster level the WaitQ is consulted to see if there is a WaitCondition pending on a message with that key. If one is found, it is de-queued and asked to handle() the message. This may or may not (depending on the contents of the message and the semantics of the WaitCondition) result in the pended Context becoming runnable.

5.8.2 Distributed Message Passing

Within Diamonds we have two different basic types of messages. First, since odl is a message passing language by design, we have language level messages that we refer to as ODLMessages. Then since Diamonds is a distributed computing environment we also have runtime level messages that we refer to as DiamondsMsgs. An ODLMessage is valid in both the non-distributed runtime, as covered in the last chapter, and in the full Diamonds distributed runtime environment. A DiamondsMsg only has meaning in the Diamonds case.

In the previous chapter we discussed ODLMessage, and their format. In this section we will discuss the format of DiamondsMsgs and how we encode ODLMessages within them.

As a prototyping and design environment for odl programs we have designed the distributed message passing scheme of Diamonds to be as simple and flexible as possible. This as in most of our design decisions, has been carefully done so as not to preclude a more optimized implementation in the future.

A DiamondsMsg within the prototype is a string of character bytes. This was chosen because a character string can be sent using a single format over a wide variety of underlying transport systems, and can be used to encode a even wider variety of information. The exact contents of a DiamondsMsg will of course vary from message to message, but there is a common set of data that is to be present in each. All DiamondsMsgs contain a *Key* field, which if not use contains a zero. We use zero to indicate the absence of a key. The basic message format is,

$$\texttt{DiamondsMsg} == Key + String$$

There are some formatting requirements made on the string portion of the message. It is assumed that the fields within the string are separated by colons (i.e. ":") and that the first field is the type of the message. The message types understood by each Agent were discussed in section 5.1.2.

One of the more challenging and interesting DiamondsMsgs from a design standpoint is the one which describes an ODLMessage. A ODLMessage is a message from one object requesting another to perform some service on its behalf. The message contains information identifying the object to perform the service, an indication of the service desired, and any arguments required by that service. All of this information must be encoded into the DiamondsMsg.

In mapping ODLMessages to a distribute environment we were presented with a fundamental design decision regarding their implementation. A common approach to this sort of RPC design is the use of *stubs* [CLNZ89]. A stub is generated at both the client and server side of the call. At the client side of the call the stub is responsible for formatting a message (marshaling) and perhaps dispatching it. At the server side the stub is responsible for un-formatting (unmarshalling) the message. In a fine-grained language such as odl the support of fine-grained objects also tends to lead to the design of fine-grained classes. That is, there are likely to be many classes within an odl program, most of which will need stubs generated for them to handle message passing. Given the stated goals of Diamonds we felt that the time needed to generate these stubs would be wasteful and inefficient.

As an alternative to the stub approach we went with a self describing message format. Stubs are needed at both ends of the client–server message path because they are the only ones who know how to properly interpret the contents of the message. We can instead transfer this knowledge to the message itself, and do away with the stubs. This is the approach we have adopted within Diamonds. This was possible because all of the fields within the ODLMessagethat is being sent are typed and the type information can be used with the DiamondsMsg to describe the ODLMessage.

Even though we have been able to do away with the myriad of stubs that would be required in a stub based messaging scheme, we still need the ability to translate from a ODLMessageto a self describing DiamondsMsg and back again at the other side of the message send. The mechanics for this translation are encapsulated within a single entity, the MsgTranslator. The MsgTranslator at the client end, takes a ODLMessage and produces a DiamondsMsg that can be sent over the network. At the other end, a different instantiation of the MsgTranslator can take a DiamondsMsg and produce an ODLMessage.

Our simple but elegant message passing scheme is another example of where we were able to push the object oriented design principles of encapsulation, localization and abstraction to produce a flexible infrastructure that could be realized in a timely and efficient manner. It is also another example of a re-occurring theme in the our design of the prototype, simplicity is a secondary rather than a primary characteristic of a good design. That is, one must carefully design abstractions to produce a simple and elegant design. Simplicity or elegance do not come without work.

5.9 Inter-Class Protocols

There are a few protocols that from a design perspective spread across our implementation. This section discusses several common design features that we have implemented throughout our prototype. It is related to Diamonds only in that our prototype is a realization of that design.

From experience, we know that maintenance is not a single phase in the life cycle of a program, but rather an activity that begins as soon as coding starts and continues throughout the lifetime of the program. To aid this activity we have required a debugPrint() method on just about all of our language related classes. This in conjunction with command line arguments to invoke it, can be used to verify the internal consistency of the objects, and to make comparison from on representation of the information to another. This sort of multi-class protocol is invaluable for debugging and maintenance.

The ability to communication options and other "program" state information in a controlled manner can always be troublesome in a design. We have done this via a Flag class and a set of subclasses from this class. The attributes of the Flag class are a set of binary and integer attributes, some of which are specified as command line arguments, other are internal defaults. The design of the Flag and its sub-classes was integral in getting the many classes within the language and runtime components of Diamonds to be reusable to the extent that they are. For the most part the Flag classes are the only expected link between one set of classes and another.

5.10 Resource Management

Resource management (RM) is crucial to effective use of any heterogeneous distributed system. Resource management services are responsible for monitoring the performance metrics of its composite resources and then based on application requests or detected imbalances seeks to reconfigure the system. All configuration decisions must comply with the current set of RM policies. Such policies address availability and reliability of system critical tasks, system security, and general system integrity. Typical load balancing algorithms[HJ86] suffer from an inability to characterize key application-based (process) performance metrics. This problem is compounded when a large number of the "processes" in the system are *subtasks* of distributed applications. Moving these subtasks, based only on resource loading, may significantly perturb the affected application's throughput and in effect, reduce system effectiveness.

The run-time composition of clusters into ensembles within Diamondspermits great flexibility in mapping an application to a variety of architectural granularities. The run-time system would like to have clusters which are large enough to be efficiently managed while at the same time are small enough to be able to exploit the concurrency available in the application and the distributed system.

Clusters are inherently more manageable when it comes to contemplating the system-wide impact of a placement decision. Clusters are of a reasonable size, both in complexity and processing activity, for migration. Cluster migration is considerably cheaper than process migration and can be used to bring tightly-coupled clusters together within a common protection domain or contra-locate loosely-coupled (highly concurrent) clusters. Diamondsis optimized for cluster migration, rather than object migration.

5.10.1 Diamonds-based Application Metrics

The central run-time abstraction is the cluster. The performance metrics which need to be considered are:

- Processing time

- Blocked time
- Intercluster communication events (and their type: synchronous/asynchronous).

For each of these metrics, the measurable quantities encapsulated within each cluster are enumerated.

Processing Time Several quantities relate to how much processing is being done by a cluster. One quantity is the *damped* average number of dispatchable *methods*. These represent queued messages which result in processing. A high number indicates the cluster is not getting enough physical processing resource, while a very low number indicates a certain degree of idleness. A second quantity, which helps relates relative processing speed, is number of methods dispatched (and approximate cycles) in the last quantum.

Blocking Time The mean dispatchable method queue length is not a complete indicator of either processing time or blocking time. A second queue within each cluster maintains the *pending contexts.* A method or context might *pend* if:

1. It is a *guarded* method and the current evaluation of the guard predicate is false.
2. It has a synchronous method invocation currently outstanding. This synchronous invocation may be "local" to the cluster or involve a remote cluster.

For determining *coupling* with other clusters, the important elements on the pend queue are ones which are blocked on a synchronous remote invocation. We will denote these as PRSI contexts (for pending on remote synchronous invocation). Of most critical concern is when the "dispatchable queue" is empty and the number of PRSI contexts is non-zero. This indicates that the "cluster" is blocked waiting on remote computations to complete.

Communication Events Coupling between clusters is reflected by joint communication events. There are four types of invocations which are visible to the cluster:

1. Synchronous intra-cluster invocation
2. Asynchronous intra-cluster invocation
3. Synchronous inter-cluster invocation
4. Asynchronous inter-cluster invocation

Asynchronous invocations result in a temporary increase in the level of concurrency while synchronous invocations represent a migration of a *thread of control.* Local synchronous invocations only suggest continued processing within the cluster, while local asynchronous invocation indicates a temporary increase in intra-cluster concurrency. Asynchronous remote invocations indicate a contribution to the remote cluster's concurrency while remote synchronous invocations indicate inter-cluster coupling.

5.10.2 Cluster Mapping

A good mapping of clusters to ensembles seeks to optimize across several parameters:

- Approximate balance of computational load across ensembles,
- Minimize inter-ensemble synchronous invocations, and
- Maximize inter-ensemble asynchronous invocations.

An application's performance profile can be fully illustrated via a weighted application performance profile graph (WAPG), with nodes representing clusters and edges representing inter-cluster communication (Fig. 23). The nodes are given a weight consistent with the cluster's mean dispatchable queue length. While each arc is labeled with two weights: 1) one indicating the rate of synchronous communication between two clusters, and 2) one for the rate of asynchronous communication. (Note this is essentially equivalent to the object graph described in Chapter 3 and in theory used to cluster objects). Mapping clusters to preexisting ensembles, corresponds to clustering the nodes of the WAPG. A best clustering would attempt to simultaneously optimize across each of the three parameters. Given this model, there is duality in minimizing or maximizing communication rates (i.e., maximizing inter-ensemble asynchronous invocation via a particular mapping is equivalent to minimizing intra-ensemble asynchronous communication rates).

A desired mapping of clusters to ensembles can be expressed via the following optimization equation:
$C = \sum_{ensembles} (\Delta_{load} + \beta \times IAEA) + \alpha \times \sum IEES$ where:

C is the total cost (to be minimized).

Δ_{load} is an indication of quality of the "load balancing".

IAEA is the intra-ensemble asynchronous invocation rate, and

IEES is the inter-ensemble synchronous invocation rates.

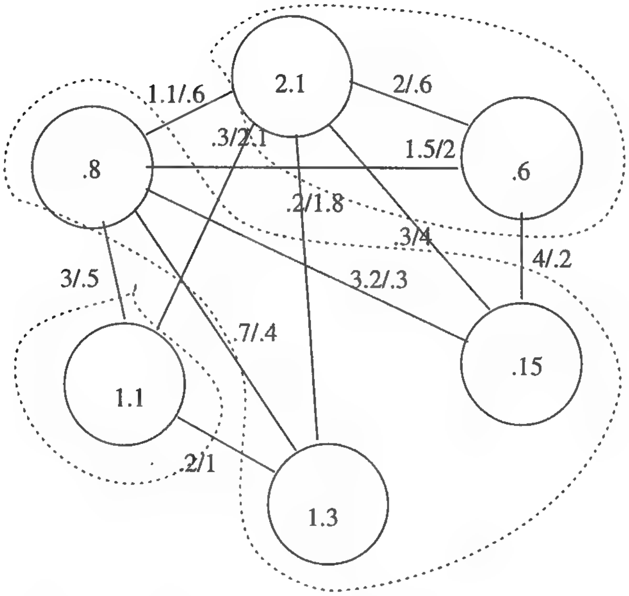

Figure 23: Weighted Application Performance Profile Graph. (Arcs labeled with Synch/Asynch communication rates).

α, β are scaling factors to adjust for sampling rates and relative magnitudes of each of the component metrics.

The Δ_{load} factor is computed first by determining the current "view" of the application's load, which is the sum of all of the individual cluster's mean length of the dispatchable methods queue. This value is then divided by the sum of the ensemble weights (total number of processors available to the application). This yields the desired "balancing" figure, i.e., the mean number of dispatchable methods per processor. For a given mapping, the actual applied load (of an ensemble) is the sum of the constituent clusters queue length divided by the weight of the ensemble. The Δ_{load} is then the difference between these two values. For example, Fig. 23 has 6 clusters with a combined load of 6.05. If we map these clusters onto 3 ensembles (each of weight 1), then the desired per ensemble load is 6.05/3 = 2.02. For the three ensembles (grouping) shown with the dotted lines, the Δ_{load}s are 2.7-2.02 = .68, 2.02-1.1 = .92, and 2.25-2.02 = .23 respectively. For the same ensemble orderings, the IAEA values are .6, .7 and 0 respectively (thus this mapping is very good at minimizing "un-tapped" concurrency). Finally, for this mapping there are 3 IEES values of .3, 7.1 and 3.2. If α and β are set to 1, the total cost for this configuration is 13.73.

The sense of balancing reflected by the Δ_{load} metric, depends greatly on our "approximate homogeneity" assumption. However, some points should be made with regard to its validity in practice. First, the model can be extended to reflect the relative throughputs or *residual capacity* of the actual resources available (being allocated) in the system. Secondly, the cost function is most frequently going to be used to suggest a "good" modification to a current configuration, i.e., given a current configuration and a goal to improve the situation (with possible changes in resource allocation) – what allocations should be changed. As indicated before, the mean dispatchable methods queue length is a reflection of "resource allocation adequacy". If the value is high (irrespective of the residual capacity of the corresponding resource), the collection of clusters are suffering starvation. Starvation is due to resource capability and *interference.* The interference may be inter-application or intra-application. True intra-application interference can be correlated with a low intra-ensemble synchronous invocation rate.

5.11 Resource Management USE OF Application Management

The cluster metrics and mapping model described above should be queried in a number of possible RM decisions. Some of these decisions are "private" to a given application, while others impact the distributed system. Consider the following *events*:

- An application needs to create a new cluster and subsequently needs to determine its best placement.

- An application detects it is in a poor configuration or inefficient resource allocation (or both).
- The system needs to change the physical mapping of an application's ensembles. This may involve a change in total resource allocation or just a reallocation of resources (as might be neccesary to provide system-wide balancing).

For the third category of events the *system* determines its need to alter an application's configuration (e.g., system load-balancing, policy driven change to application's resource allocation, etc.). It is desirable that it does so in an *application friendly* way, i.e., considering the application's current performance profile.

What is interesting about the proposed architecture is the second group of events. In these cases, the application senses bottlenecks which may be alleviated by a RM activity.

5.11.1 Application Performance Metrics Interface

Both an application and the RM services need a way to efficiently extract from an application a set of measures which reflect the current computation and communication interaction among its clusters. For an application with hundreds of clusters, this interface cannot require the sequential query of each cluster. The dissemination of performance metrics must be accomplished in a hierarchical manner using the existing distributed application tree.

An application is likely to be comprised of fewer than a hundred ensembles (and frequently on the order of tens of ensembles). The ensembles cooperate to maintain an applications performance metrics. (The ensembles cooperate with their constituent clusters to collect the raw metrics.) These "raw" metrics include:

- Number of messages(contexts) in each of the following "queues"

  Firable Methods – Viewed as runnable threads

  Pending Contexts – Those blocked on a guard condition

  Blocked Contexts – Those threads waiting on completion of a synchronous request.

- Total count of both messages received (external) and number of contexts initiated.
- Count of both inter-cluster synchronous and asynchronous invocations – by target cluster..

These raw counts must be converted to either running average (damped value) or a damped rate. This requires periodic "real-time" scanning by the ensembles. The ensembles coordinate to collect a complete set of application metrics, i.e., the WAPG. Given this WAPG and a current weighted number of ensembles, it is possible to apply a partitioning/clustering algorithm which seeks to minimize the cost function defined in section 5.10.2. An interface is presented to the resource management facility permiting a single query to any ensemble. The protocol results in message forwarding to all other application ensembles. Each (including the original) respond directly to the resource management facility with their subset of performance metrics (they also reply with application id and node id).

For the application self-monitoring, the ensembles will periodically multicast a summary of their performance metrics to all other ensembles associated with the same application. Upon receipt of these metrics, each ensemble will determine the application's average values for "loading", inter-cluster communication rates (both synchronous and asynchronous), IAEA, and IEES. Heuristic thresholds are used to predict when intervention by the resource management system is likely to yield an application configuration change which could significantly improve performance. An ensemble can autonomously detect the following situations:

- Ensemble load significantly different from the application nominal load.
- Higher the nominal inter-ensemble synchronous communication rate (particularly if contributed by a select few clusters).
- A higher than nominal intra-ensemble asynchronous communication rate.
- A partial WAPG exists solely within each ensemble. This can be evaluated to determine if the collection of clusters form one or more tightly coupled subsets. The presence of multiple sets indicate a lack of cohesion in the ensemble (especially, when there is a high asynchronous communication rate between such subsets).

In simulation studies, a particularly effective heuristic threshold has been discovered which is the product of two communication based ratios (ARIE * SREI):

ARIE – the local mean per-cluster inter-cluster asynchronous communciation rate (which is intra-ensemble for the application) divided by the global mean per-cluster inter-ensemble asynchronous communication rate.

SREI – the gloabl mean per-cluster inter-ensemble synchronous communication rate divided by the local mean per-cluster inter-cluster synchronous communication (local, intra-ensemble).

The above metric and a threshold of .75 was used to discriminate between good and bad cluster placement for 2500 random WAPGs. For each "perturbed" WAPG, a "good" placement was computed for comparison. In the table below, the descriminating ability of the metric is described for sets of WAPGs grouped by "how bad" the set of random graphs were (percent difference in cost function between perturbed WAPG and the "good" placement). As the configuration "worsens", the probability that the metric will declare this fact increases. The first line show the number of times the metric also declares the "good" configurations as bad!

Condition type	prediction/ events	hit rate	percent tests
False detects	182/2500		7.3%
2-6%	153/400	38.3%	16%
6-12%	443/710	62.4%	28.4%
12-18%	430/536	80.2%	21.4%
>18%	392/473	82.9%	18.9%

Table 9: Discrimination Ability of Heuristic Metric

Resource management and application configuration management are two challenging problems within distributed systems. Both problems share common goals while they both place constraints on the other. For most distributed software frameworks, very little application performance information is available (for either application configuration management or resource management). Diamonds has a unique architecture which exposes significant application performance metrics.

Chapter 6

Current Status and Future Directions

As part of the completion of this project, three prototypes of various parts of DIAMONDS is being delivered. An ongoing activity within the CASE Center is the refinement and extensions of these prototypes into a more complete system. This is being done principly to support current and experimental research in OO distributed systems.

The three prototypes are:

- ODL to SIRF translator. This prototype supports about 85% of ODL as described in Chapter 2.
- Distributed Run-Time. This prototype of the drt runs on top of Unix and utilizes Unix sockets for communication. It does not have any provision for cluster migration or intelligent resource management. It also currently only supports *asseed* and *with*[1] hints for object placement.
- Standalone static clustering tool + simulator. This prototype builds off a common base as the internal portion of the translator described above. It employs the heuristics described in Chapter 4. A simple simulator which only considers mapping of clusters to processors (i.e., no ensembles) – then evaluates the validation metrics also outlined at the end of Chapter 4.

The current plans are for an integrated public release of these tools in September 1994. In that release, it is anticipated that the clustering tool will be integrated with the translator. The distributed run-time should fully support (in a naive manner) all of the object placement hints

[1] A restricted form of *closeto* with a single object reference to guide placement.

provided by the clustering tool. Additional feature which hopefully will be partially supported in this release include:

- primitive resource manager to aid in initial configuration (mapping ensembles to processors).
- a monitoring tool to track either in real-time or postmortem performance metrics of an application.

Research in OO Method and Tools and in DIAMONDS has evolved through a close coupling between theory and practice. A major component of the work would best be described as Experimental Computer Science and Engineering. The current DIAMONDS framework begs for significant refinements and extensions as well as a significant set of performance evaluations and characterizations. In addition there are many interesting topics which may have a more foundational impact on our work. Thus, a description of future efforts by this group is roughly separated along those lines.

6.1 Future Development

One of the most profound challenges for distributed software engineering is to be able to balance ease of development (flexibility) with the underlying need for performance. Much of the future work within the DIAMONDS framework will push for performance optimizations. These include:

- Mapping of the run-time to microkernel based infrastructure.
- Development of a refined resource management facility to support application-based dynamic placement (cluster and ensemble migration).
- A dynamic compilation environment to produce cluster-specific native-code on a demand basis.
- A redeployment of a DIAMONDS simulation environment is also planned.

Each of these are examined more closely in the following sections.

6.1.1 Mach-based DIAMONDS run-time

Modern day microkernel technology offers a number of abstractions which more directly support the DIAMONDS run-time. This is not entirely by coincidence as DIAMONDS has been designed with this infrastructure in mind. Some of the more relevant mappings include:

- Ensembles map to tasks, while clusters associate with abstract memory objects.

- A MachIPCjunction will manage a port, and each agent (be it cluster, ensemble, or program) will be associated with a port. Ports rights for remote clusters and ensembles are managed by each Ensemble.

- Threads and External pagers will be exploited where they make sense. C-Threads in Mach are still relatively expensive and will not likely replace our notion of Contexts.

6.1.2 Resource Management

A design and protocol to monitor an application's performance metrics is complete. The near-term issues are to implement the coordination between ensembles and clusters to collect and assemble the metrics. Then the coordination between an application's ensembles will be implemented and tested. An external monitor will then periodically query an applications profile and display it. The next step will be to couple application monitoring with physical resource monitoring[2] and build an intelligent resource scheduler.

The next phase will be to finalize the cluster migration protocol within the run-time and then build a general purpose cluster migration server. Finally, the cluster migration server, application monitor and the resource manager will coordinate to control distributed applications within DIAMONDS.

6.1.3 Dynamic Compilation

As part of [Sha94], a framework was established to provide for various techniques to optimize code execution within DIAMONDS. Key to this is support for dynamic compilation. This permits the production of native code (as opposed to SIRF code). We have chosen the dynamic compilation route for two main reasons:

1. Our desire to support heterogeneous environments while at the same time supporting rapid prototyping pushes us away from static compilation.

2. Many of the optimizations available to active object lanaguages, such as message inlining, message splitting, customization, etc., require locality information. (Applying these optimizations on remote invocations paths will yield little perceptible speed-up while application to local, intra-cluster invocations should be significant.) An object's relative locality is only known upon construction (after it has been placed within a specific cluster). Thus, compiler optimizations are best applied when this locality information is available.

[2] We have a number of such environments built at Syracuse from which we can reuse.

The run-time will be augmented to support the binding to compilation agents and a *code hierarchy* consisting of multiple cache levels will be implemented to permit a blend of interpretation, general machine-specific binaries, and cluster specific binaries to coexist within an application.

6.1.4 Simulator

The current simulator is tightly integrated with the parser and clustering tool. In fact the clustering tool and simulator run off of the ast representation rather than sirf. A more complete simulator, whose principle purpose is to permit extensive instrumentation and tracing, is currently being designed and implemented. It will run off of sirf code, and will simulate all aspects of the DIAMONDS run-time. It will permit plug replacement for different "algorithms" for such things as cluster migration, resource management, communication protocols, context scheduling, etc. It will also permit the ability to simulate different hardware architectures (both processors and interconnection architectures).

6.2 Future Research

As indicated before some of the future work will be more foundational in nature. These include:

- Further refinement of the static clustering models and heuristics.
- Support for real-time constraints with both odl and the run-time.
- Design constructs and run-time support of group-based designs for reliable systems.
- Support for persistent clusters, including issues of naming, activating, and sharing. Of particular interest is support for multimedia objects via DIAMONDS clusters.
- Communication protocol work. DIAMONDS primitive communication is NOT request-response. Protocols to support efficient streaming and negative acknowledgements need to be established.

6.2.1 Static Clustering

The original model established for clusting objects only considered interactions between objects which indicated that they should colocate. This was abandoned when a heuristic tool was developed since a default action of placing objects into their own cluster (given no useful interaction information) is not pragmatic. The heuristic approach concentrated on class-based affinities between an object being constructed and objects with which ig may potentially interact (limited to those object references available at the scope of object construction).

The original model needs to be refined and the current infrastructure to compute class-based affinities updated to reflect these changes. Further, there is a large set of heuristics which need to be analyzed for their potential effectiveness within the "pseudo-static" clustering framework of DIAMONDS. In addition to the annotations to the "new" operator, mechanism for passing static analysis information on to the run-time system need to be considered.

6.2.2 Real-Time Active Objects

Active objects have a close kinship with reactive systems. Currently however, odl provides no explicit syntax for describing real-time constraints. Many real-time languages and systems establish real-time control constraints within their *thread* abstraction. DIAMONDS has chosen a *thread-less* model. In order to provide real-time support, this group needs to explore modeling techniques such as employed by [JKS91]. Additionally, the run-time system must provide a means for tagging messages with scheduling constraints.

6.2.3 Objects in Groups

Objects, as encapsulations of services, are ideally suited for specifying reliable distributed services through replication. The most common replication techniques involve some sort of groups [PCD91, SDP91, Bir93]. We have worked on a preliminary framework for supporting groups within object-oriented models. Future work will provide design support within odl, the clustering tool, and the run-time system.

6.2.4 Persistent Objects

Many applications require the support of persistent objects. Most notable are transaction processing systems and multimedia applications. Many of the characteristics which make clusters efficient for virtual memory subsystems and migration make them an ideal abstraction for persistence. Persistent clusters in general will have different semantics, in that they will be slowly changing or perhaps even static. Such properties need to be exploited.

Designers need a way to express that a cluster is a prescribed entity with an extended lifetime. Also, the run-time system must be adapted to support the naming and management of these persistent clusters. Finally, as outlined in Chapter 5, the run-time must support *cluster-faults* and the necessary mechanism to locate and activate inactive persistent clusters.

6.2.5 Communication Protocols

odl supports both synchronous (bidirectional) communication as well as asynchronous (unidirectional) communication. The current translator and run-time translates all non-local communications into "one-way" messages. This is advantageous for a number of reasons, but most notably it no longer requires request-response services from the communications layer. Appropriate communication protocols need to be developed which can effectively piggy-back messages between ensembles and minimize the occurence of acknowledgement messages while maintaining reliable delivery semantics. This should be of greatest interest to provide DIAMONDS application protocols on top of ATM networks.

Bibliography

[ATK92] A. L. Ananda, B. H. Tay, and E. K. Koh. A Survey of Asynchronous Remote Procedure Calls. *Operating Systems Review*, 26(2):92–109, April 1992.

[Bir85] A. Birrell. Secure Communication Using the Remote Procedure Call. *ACM Transactions on Computer Systems*, 3(1), 1985.

[Bir93] K. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12):36–53, December 1993.

[Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin Cummings, 2nd edition, 1994.

[CBHS93] Vinny Cahill, Shean Baker, Chris Horn, and Gradimir Starovic. The Amadeaus GRT - Generic Runtime Support for Distributed Persistent Programming. In *OOPSLA '93*, pages 144–161, 1993.

[Cea94] D. Coleman and et. al. *Object-Oriented Development: The Fusion Method.* Prentice-Hall, 1994.

[CGZ94] Brad Calder, Dirk Grunwald, and Benjamin Zorn. Quantifying Behavioral Differences Between C and C++ Programs. Technical Report CU-CS-698-94, Department of Computer Science, University of Colorado, 1994.

[CKP93] Andrew Chien, Vijay Karamcheti, and John Plevyak. The Concert System – Compiler and Runtime Support for Efficient, Fine-Grained Concurrent Object-Oriented Programs . DCS Technical Report UIUCDCS-R-93-1815, University of Illinois, Department of Computer Science, 1304 W. Springfield Avenue, Urbana, Illinois, June 1993.

[CLNZ89] S. K. Chung, E. D. Lazowska, D. Notkin, and John Zahorjan. Performance implications of design alternatives for remote procedure call stubs. In *The 9th International Conference on Distributed Computing Systems*, pages 36-41, Newport Beach, CA USA, June 1989. IEEE.

[Cor90] IBM Corporation. *POWER Processor Architecture.* Advanced Workstation Division, Austin, TX, 1990.

[CR92] Arunodaya Chatterjee and William A. Rogers. Integrated Resouce Allocation in ESP. Technical Report ESL-ESP-194-92, MCC, October 1992.

[CUL89] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of SELF – a dynamically-typed object-oriented language based on prototypes. In *Proceedings OOPSLA '89*, pages 49–70, October 1989. Published as ACM SIGPLAN Notices, volume 24, number 10.

[dCLF93] Dennis de Champeaux, Doug Lea, and Penelope Faure. *Object-Oriented Software Development.* Addison-Wesley, 1993.

[Dic91] Peter Dickman. Effective Load Balancing in a Distributed Object-Support Operating System. In *1991 Workshop on Object Orientation in Operating Systems*, 1991.

[Dra90] Richard P. Draves. The Revised IPC Interface. In *Proceedings of the USENIX Mach Conference*, October 1990.

[Gol83] Adele Goldberg. *Smalltalk 80: The Language and its Implementation.* Addison-Wesley, 1983.

[HJ86] Anna Hac and Theodore J. Johnson. A Study of Dynamic Load Balancing in a Distributed System. In *SIGCOMM Sym. on Communications Architectures and Protocols*, pages 348–356, August 1986.

[HK87] C. Horn and S. Krakowiak. Object Oriented Architecture for Distributed Office Systems. In *Procedings of Esprit Conference*, 1987.

[Inc87] Sun Microsystems Inc. *The SPARC Architecture Manual.* Mountain View, CA, 1987.

[Jea92] I. Jacobson and et. al. *Object-Oriented Software Engineering.* Addison-Wesley, 1992.

[JKS91] H.-M. Jarvinen and R. Kurki-Suonio. DisCo Specification Language: Marriage of Actions and Objects. In *Proceedings 11th International Conference on Distributed Computing Systems*, 1991.

[Joh86] Stephen C. Johnson. Yacc: Yet Another Compiler-Compiler. In *Unix Programmer's Supplementary Documents, Volume 1 (PS1)*, chapter PS1:15. USENIX Association, 1986.

[Lea91] Doug Lea. Steps Toward an Internal Representation System for the Semantic Analysis of C++ Programs. In *Proc. of OOPSLA*, October 1991.

[Lea94] D. Lea. ODL: Preliminary Language Report. Technical report, CASE Center, Syracuse University, 1994.

[LFJ+86] Samuel J. Leffler, Robert S. Fabry, William N. Joy, Hil Plapsly, Steve Miller, and Chris Torek. An Advanced 4.3BSD Interprocess Communication Tutorial. In *Unix Programmer's Supplementary Documents, Volume 1 (PS1)*, chapter PS1:8. USENIX Association, 1986.

[LS86] M. E. Lesk and E. Schmidt. Lex - A Lexical Analyzer Generator. In *Unix Programmer's Supplementary Documents, Volume 1 (PS1)*, chapter PS1:16. USENIX Association, 1986.

[PCD91] D. Powell, M. Chereque, and D. Drackley. Fault-Tolerance in Delta-4. *ACM Operating Systems Review*, 25(2):122–125, April 1991.

[Rea91] J. Rumbaugh and et. al. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[Ros91] Jim Roskind. `c++grammar2.0`. The source for this grammar is available at USENET archive sites for `comp.lang.c++`, 1991.

[SDP91] S. Shrivastava, G. Dixon, and G. Parrington. An Overview of the Arjuna Distributed Programming System. *IEEE Software*, pages 66–73, January 1991.

[Sha94] C. Kevin Shank. *A Computing Framework for Open Distributed Systems.* PhD thesis, Syracuse University, 1994.

[Sta84] James W. Stamos. Static Grouping of Small Objects to Enhance Performance of a Paged Virtual Memory. *ACM Transactions on Computer Systems*, 2(2), May 1984.

Appendix A

ODL EBNF

The following EBNF syntax uses “[...]” for “optional” and “[...]*” for “zero or more”.

```
System:        [ Decl ]*
Decl:          Class | Fn | Op | Inv | Init | Open | Gen | Locals | Accept | Rec | ;
Class:         class GID [ is GIDs ] [ Decl ]* end
Fn:            [ local | own | packed ] [ fn ] GID Params : QualType FnDef
Op:            [ local ] op GID Params ReturnSpec OpDef
Inv:           inv Exps
Init:          init Exps
Open:          opens GID
Gen:           generator GID
Rec:           record GID Params
Locals:        locals [ Decl ]* end
Accept:        when Exp then [ Op ]* ElseAccepts end
ElseAccepts:   [ elsewhen Exp then [ Op ]* ]* else [ Op ]*
Params:        [ ( ParamList ) ]
ParamList:     GID : QualType [ , GID : QualType ]*
QualType:      [ fixed | unique | common | opt ]* GID
ReturnSpec:    [ [ ID ] : QualType | : Synch [ , Synch ]* ]
Synch:         [ ID ] ( [ ParamList ] )
FnDef:         [ [ init ] = Exp ] FnBind
FnBind:        <> | Block | ;
OpDef:         Block | Effect | ;
Effect:        ==> OpSpec end | When
```

```
When:         when Exp then OpSpec ElseWhens end
ElseWhens:    [ elsewhen Exp then OpSpec ]* else OpSpec
OpSpec:       [ When | Exps [ Block ] | Block ]
Block:        { Statements }
Statements:   Statement [ ; Statement ]*
Statement:    [ Exp | Assign | Loc | Catch | While | If | Reply ]
Reply:        reply [ Exp ]
While:        while Exp do Statements end
If:           if Exp then Statements ElsIfs end
ElsIfs:       [ elsif Exp then Statements ]* [ else Statements ]
Catch:        catch Exp [ Op ]* end
Assigns:      Assign [ , Assign ]*
Assign:       GID := Exp
Loc:          local GID : QualType [ := Exp ]
Exps:         Exp | Exp , Exps
Exp:          [ @ ] Exp2
Exp2:         [ Exp2 OrOp ] Exp3
OrOp:         \/ | =>
Exp3:         [ Exp3 /\ ] Exp4
Exp4:         [ Exp5 RelOp ] Exp5
RelOp:        = | < | > | ~= | >= | <=
Exp5:         [ Exp5 AddOp ] Exp6
AddOp:        + | -
Exp6:         [ Exp6 MulOp ] Exp7
MulOp:        * | / | div | mod
Exp7:         [ Unop ]* Exp8
Unop:         - | ~
Exp8:         PredefFn | PredefExp | Msg | ( Exp )
PredefFn:     Msg in GID | null ( Msg ) | oneOf ( GIDs )
PredefExp:    true | false | null | pend | literal
Msg:          Rcvr [ . Send ]* [ ' | '' | ? ]
Rcvr:         self | [ GID $ ] Send | new GID [ ( [ Assigns|Exp ] ) ]
Send:         GID [ ( [ Exps ] ) ]
GID:          ID | GID [ Exps ] | PredefType
PredefType:   bool | int | char | real | time | blob | Any | System
GIDs:         GID [ , GID ]*
ID:           [ ID : : ]* name
```

Appendix B

Building Diamonds

B.1 The Diamonds source structure

In the root directory are,
Some `.readme` files,

`BUILD.readme` : How to build Diamonds

`STATUS.readme` : Notes on the current status of the Diamonds tools

`USAGE.readme` : How to run the Diamonds programs

some subdirectories,

```
      bin - This is where final executable go
      lib - This is where we place libraries
      obj - The ".o" files go here
      src - The code is down in this subdirectory
```

and

```
      odlCode - Simple examples. Most are parser test
                files that don't do anything interesting,
                but instead test some particular feature
                of the language.
      sirfCode- The sirf code generated from the
```

```
          "odlCode" files
scripts - Some utility perl scripts
          (more on these later)
```

Diamonds is targeted towards a heterogeneous environment of machines. To support this we have architecturally specific subdirectories in each of `bin`, `lib`, and `obj`. At the moment those subdirectories are,

```
aix_rs6000
hpux_snake
osf1_i386
sun4_sparc
```

With this setup and the organization of the Makefiles (as will be discussed below) we can build Diamonds for several different architectures from the same source tree.

In the src directory are,
the following subdirectories,

```
ast          - The Abstract Syntax Tree classes
               for ODL
aux          - Some Auxiliary classes used
               throughout Diamonds
commUtils    - A set of Communications Classes
diamonds     - The Diamonds Runtime Classes
distRT       - The "driver" program for drt
grammar      - The ODL grammar files
               (yacc and lex type)
interpreter  - The "driver" program for interp
main         - The "driver" program for "_odl"
parserRep    - Classes to support a parsed
               representation of an ODL
               program
rmtEnsemble  - The "driver" program for "ensemble"
runTime      - Generic runtime classes.
simulator    - The SIRF Simulator
sirf         - The SIRF code classes.
```

some makefiles,

```
Makefile
Makefile-distRT
Makefile-ensemble
Makefile-interpreter
Makefile-odl
```

and one header file,

```
odl.defs.h
```

B.2 Building Diamonds

Diamonds is made up of a set of class libraries. Many of these classes are used throughout the different programs that comprise the Diamonds system. To make the build process a bit more coherent and flexible and also to support this sort of internal reuse, each "directory" under /diamonds/system/src, is composed of all the files necessary to create a library file of the same name. So, the directory src/ast when compiled will produce a libast.a as a result. This means that if you are going to be working with only one or two directories within the src tree, you can make use of pre-existing Diamonds libraries to speed up your compiles.

The Makefiles within Diamonds are structured to allow a flexible arrangement of target directories. This allows disk space intensive files like ".o"s, libraries, and compiled binaries to be moved to "scratch" space.

To support this arrangement, the Makefiles will look for a user defined variable DiamondsRoot. DiamondsRoot specifies the complete path to a directory which should contain the following directories,

```
bin
obj
lib
```

The DiamondsRoot can be on any mounted disk that is readable and writable by you. Each of the subdirectories bin, obj and lib in turn have their own subdirectories. Since all of this can get a little bit confusing there is a Perl script called makeDiamondsTree available to help set up the proper files and directories under the DiamondsRoot. To use makeDiamondsTree, first create the directory that will be the the root of the Diamonds compile tree. For example let's say this is /scratch/myDiamonds. This first step would be,

```
mkdir /scratch/myDiamonds
```

then the makeDiamondsTree script must be run. This script can be found in /diamonds/system/scripts As an argument the script takes the name of the directory that is to be the DiamondsRoot. So to run this script and build all of the subdirectories, you can type (continuing with our example),

```
/diamonds/system/scripts/makeDiamondsTree /scratch/myDiamonds
```

There are just a few more more steps now.

First, chose a directory for your copy of the Diamonds source tree. For our example case, this might be /scratch/myDiamonds/src. The next step is to copy the original source files to this directory. In our example we could do this by typing

```
cp -R /diamonds/system/src /scratch/myDiamonds
```

Now you need to define the DiamondsRoot variable. This is just the way to let make know where we have located our version of DiamondsRoot, so you can do this at your shell prompt by typing,

```
export DiamondsRoot="/scratch/myDiamonds"
```

or,

```
setenv DiamondsRoot "/scratch/myDiamonds"
```

depending on your shell.

Finally, you also need to indicate which type of machine you are going to be compiling this on. There are currently four options,

- Sun Sparcstations
- HP 700 Series Workstations (aka Snakes)
- IBM RS6000 Workstations and
- Intel based x86 Machines

To indicate to make which machine you are targeting you must again define a variable. In this case it is DiamondsTarget.

Machine	DiamondsTarget
Sun	sun4_sparc
HP	hpux_snake
IBM	aix_rs6000
Intel	osf1_i386

So figure out what machine you are currently running on, and then define the proper DiamondsTarget. If it is a sun sparcstation this would be done by typing,

```
        export DiamondsTarget="sun4_sparc"
```

or,

```
        setenv DiamondsTarget "sun4_sparc"
```

(assuming we are running SunOS 4, which we are in the Cat Cluster).

Now you can start compiling all of Diamonds by cd'ing to the src directory and typing make.

If you type make with no arguments it will take about an hour or so on the SparcStations to rebuild Diamonds from scratch. If you only want to build the "language" portion of it (No Diamonds stuff), you can type,

```
        make lang
```

If you only want the Distributed Runtime environment you can type,

```
        make dist
```

Appendix C

Using Diamonds

At the moment there are three programs that can be invoked from the command line.

`odl` : The parser, and SIRF code generator

`interp` : A SIRF code interpreter (pure odl no Diamonds stuff)

`drt` : The Diamonds runtime environment for executing sirf files.

In the next three section we summarize their use, and describe any particular “features” that a user should be aware of.

C.1 odl – The ODL parser and SIRF code generator

```
Usage: odl <flags> <odlFile>
  where <odlFile> is the odl code
  and <flags> is any of

        -v : print version number
        -c : Show Classes
        -i : Print instructions during simulation (sets -s too)
        -m : Show method code (sets -c too)
        -r : Print registers during simulation (sets -i -s too)
        -s : Simulate sirf code
        -v : Print version bulletin
        -w : Write sirf to file
        -x : Show sirf
        -z : Show Context Scheduling info
        defaults (w) : Use caps to override defaults (i.e. -E)
```

the flags can be in any order with the odlFile, but must be separate, i.e. -v -s not -vs.

odl is actually a Perl script that will call the actual parser, _odl, for you. This is because, at the present time, odl programs are preprocessed with cpp in order to include library and other files. To assist this process there is a Perl script odl that acts as a front end for the real _odl program. This Perl script does a few nice things like look in your environment for several variables, namely

```
odlRoot : the directory where _odl is.
odlSourceDir : a directory where you
               have many odl programs.
```

The odl script assumes that the file extension of a odl program is .odl so if you say,

```
odl foo
```

it will look for

1. foo in the current directory

2. foo in your odlSourceDir directory

3. foo.odl in the current directory

4. foo.odl in your odlSourceDir directory

if it cannot find foo it will let you know.
So, to use odl, do the following.

1. Set odlRoot to the directory where the _odl is that you want to run.

   ```
   ex.   export odlRoot="/diamonds/system/bin/sun4_sparc"
   or    setenv odlRoot "/diamonds/system/bin/sun4_sparc"
   ```

   depending on your flavor of shell.

2. If you have a directory where you will be working on odl files, you can take advantage of the odlSource environment variable. If you don't specify this variable, you will need to always provide a complete path to the file that you want to translate.

   ```
   ex. export odlSourceDir="$HOME/odl/files";
   ```

3. Make sure the odl Perl script is in your path,

   ```
   ex. export PATH="$PATH:/diamonds/system/scripts"
   ```

4. run the program

   ```
   odl <odlFile> <flags>
   ```

C.2 interp – The SIRF Interpreter

```
Usage: interp <flags> <sirfFile>
  where <sirfFile> is the sirf code generated by
  the odl compiler and <flags> is any of

         -v : print version number
         -i : Print instructions during simulation
         -r : Print registers during simulation
         -s : Simulate sirf code
         -v : Print version bulletin
         -x : Show sirf
         -z : Show Context Scheduling info
         defaults (s) : Use caps to overide defaults (i.e. -E)
```

interp is a pure odl interpreter in that it does not do any Diamonds related activities. There are no clusters or ensembles. All objects reside and "execute" in a single address space.

At the present time there are no I/O primitives in odl. Until I/O is added you can check to see what is going on within your program by using the -i or -r options. These will show what SIRF instruction is executing and what registers are used, respectively.

C.3 drt – The Diamonds Runtime Environment

```
usage: drt [-?|--?]
           [-a|--autoStartEnsembles]
           [-A|--noAutoStartEnsembles]
           [-c|--clusters <int>]
           [-d|--showHandleDispatch]
           [-e|--ensembles <int>]
           [-h|--help]
           [-i|--instructions]
           [-l|--executeLocally]
           [-m|--showDiamondsMessages]
           [-p|--pauseAfterEnsembleCreation]
           [-r|--registers]
           [-s|--simulate]
           [-S|--No_simulation]
           [-v|--version]
           [-z|--ShowContextScheduling]
           <sirfFile>

         -a : Automatically start remote Ensembles
         -d : Show message dispatches
         -i : Print instructions during simulation
         -l : Execute locally (no remote ensembles)
         -m : Show Diamonds Msg traffic
         -r : Print registers during simulation (sets -i too)
         -s : Simulate sirf code
         -v : Print version bulletin
         -z : Show Context Scheduling info
         defaults (aLs) : Use opposite case to
                          overide defaults (eg. -A)
```

The default number of ensembles is 1 and the default number of clusters is 1. To get more interesting behavior you can increase these by using the -e and -c options to specify different numbers.

The -l option can be used to force a completely local execution. In this case all ensembles and clusters will be in a single address space. This can be nice for debugging.

In a remote execution, ensembles are forked and run on the same machine as the drt program. If a -A option is given, you can take over the responsibility for choosing the machine to start a ensemble, and to actually start it. If this is done, the program will request that you run the program ensemble with a particular set of arguments. To do this just find a machine and type ensemble with the EXACT same arguments as printed. For example, drt might issue the following request,

```
Please manually start:
        ensemble
with args:
        lynx.cat.syr.edu 3109 0 A
```

So you would need to find a machine, let's say "jaguar" and then type

```
ensemble lynx.cat.syr.edu 3109 0 A
```

at the command line.

When drt is run without the -l option each ensemble will open its own xterm window. Any debugging or other information from that ensemble will be printed in that window. When the program completes, the windows will remain. This allows you to see the output, or other information. To remove the windows, just select "Close" from the menu or type a "Control-C" in the window.

☆U.S. GOVERNMENT PRINTING OFFICE: 1997-509-127-61001

MISSION OF ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

a. Conducts vigorous research, development and test programs in all applicable technologies;

b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;

c. Provides a full range of technical support to Air Force Material Command product centers and other Air Force organizations;

d. Promotes transfer of technology to the private sector;

e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.